5-2012

CogTool-Helper: Leveraging GUI Functional Testing Tools to Generate Predictive Human Performance Models

Amanda Swearngin

COGTOOL-HELPER: LEVERAGING GUI FUNCTIONAL TESTING TOOLS TO GENERATE PREDICTIVE HUMAN PERFORMANCE MODELS

by

Amanda Swearngin

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Myra B. Cohen

Lincoln, Nebraska

May, 2012

COGTOOL-HELPER: LEVERAGING GUI FUNCTIONAL TESTING TOOLS TO GENERATE PREDICTIVE HUMAN PERFORMANCE MODELS

Amanda Swearngin, M. S.

University of Nebraska, 2012

Adviser: Myra B. Cohen

Numerous tools and techniques for human performance modeling have been introduced in the field of human-computer interaction. With such tools comes the ability to model legacy applications. Models can be used to compare design ideas to existing applications, or to evaluate products against those of competitors. One such modeling tool, CogTool, allows user interface designers and analysts to mock up design ideas, demonstrate tasks, and obtain human performance predictions for those tasks. This is one step towards a simple and complete analysis process, but it still requires a large amount of manual work. Graphical user interface (GUI) testing tools are orthogonal in that they provide automated model extraction of interfaces, methods for test case generation, and test case automation; however, the resulting test cases may not mimic tasks as they are performed by experienced users.

In this thesis, we present CogTool-Helper, a tool that merges automated GUI testing with human performance modeling. It utilizes techniques from GUI testing to automatically create CogTool storyboards and models. We have designed an algorithm to find alternative methods for performing the same task so that the UI designer or analyst can study how a user might interact with the system beyond what they have specified. We have also implemented an approach to generate functional test cases that perform tasks in a way that mimics the user. We evaluate the feasibility of our approach in a human performance regression testing scenario in LibreOffice, and show how CogTool-Helper enhances the UI designer's analysis process. Not only

do the generated designs remove the need for manual design construction, but the resulting data allows new analyses that were previously not possible.

DEDICATION

This thesis is dedicated to my mom and dad.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Myra Cohen, for her support and dedication to this work, and for her encouragement throughout this process. I would like to thank Bonnie John and Rachel Bellamy from IBM for being such wonderful collaborators and for teaching me so much. Also, thank-you to Don Morrison for promptly fixing CogTool issues that I had. I'd also like to thank Atif Memon for providing the GUITAR framework and technical support.

I would also like to thank Anita Sarma and Gregg Rothermel for serving on my committee and offer your thoughtful comments and feedback. To my officemates and lab-mates especially Katie, Charlie, Brady, Javier, Elena, Josh, Corey, Sandeep, Lucy, Supat, Kaylei and everyone else in the ESQuaReD lab, you have been wonderful and provided me with a lot of inspiration, fun, and laughs.

To all of the wonderful CSE professors at UNL I have had the privilege of taking classes from, you have been wonderful. I would especially like to thank Berthe Chouiery for providing me with so much inspiration and support, Ashok Samal for writing me a recommendation letter for the Anita Borg scholarship, and Charles Riedesel for being a wonderful undergraduate advisor and advisor to UPE.

To mom and dad, for everything you have given me throughout the past two years, and my whole life, and for making me the person that I am today so that I could achieve this degree. Thanks for giving me the brains! To Stephanie and Elizabeth, thank-you for being there for me throughout this process and for being my two best friends. To all of my good friends, who have helped me to make it through the past two years of work, thanks. Finally, thank-you to Will for being there for me and keeping me from going insane throughout this process, making me delicious food, and everything else. Thanks!

This work was supported in part by the National Science Foundation through award CCF-0747009 and CNS-0855139, and by the Air Force Office of Scientific Research, award FA9550-10-1-406

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

User-interface (UI) design is the process of designing computers, software applications, mobile devices, and websites with a focus on a user's interaction and experience with the system. The goal is to make the user's interaction with the system as simple and efficient as possible to help the user accomplish their goals. This is called *user-centered design*. A good interface design facilitates the user accomplishing the task at hand as easily as possible without drawing attention to itself. The process of UI design can begin with the UI designer drawing digital or paper sketches of their proposed system and performing usability testing with potential users. During usability testing, users perform tasks using the interface while UI designers observe their interactions with the system and measure their performance for tasks. In many cases, obtaining users for such studies is difficult and conducting such studies is time consuming. User-centered design is an important step in the development of a new software application, but it is very difficult. As a consequence, this step often gets cut short or left out of the development process entirely.

Techniques for *predictive human-performance modeling*, the process of modeling user tasks on an interface and obtaining performance predictions, are often used

during usability testing. GOMS is a human performance modeling technique for analyzing the complexity of interactive systems. In GOMS, a task is defined in terms of Goals, Operators, Methods, and Selection Rules. The simplest version of GOMS modeling can be done using the Keystroke-Level-Model (KLM) [9]. Developing such models consists of describing the system in different forms that are usually dictated by the modeling framework. For some, as in GLEAN (GOMS Language Evaluation and Analysis) [20], this is done through programming; for others, as in CogTool [18], a tool that allows UI designers to construct predictive models in the form of a KLM, this is done through capturing screens and drawing widgets to make a storyboard. In CogTool, designers build a storyboard representing tasks on their UI and create a cognitive model of an end-user performing a task using the Keystroke-Level-Model (KLM) [9]. The KLM runs on top of the ACT-R cognitive architecture [5] producing a quantitative prediction of performance for a skilled user. CogTool does not predict human performance for non-skilled users, however, recent work has been done to predict exploration behavior for novice users ([48]). The described techniques, CogTool and GLEAN, are useful; however, they can be very time-consuming. One approach to solve this problem is VisMap [3]. VisMap uses image processing to "see" the screen of a UI and passes the information to a human performance model that simulates motor movements (clicks, key presses) by manipulating the event queue at the operating system level. However, VisMap does not address the need to describe the tasks to be modeled.

Human performance modeling was originally conceived as an aid to design [10]. In practice, however, modeling is done just as often on legacy systems. Sometimes this is done as an analysis of existing problems; sometimes as a benchmark against which new designs are compared [7]. Modeling in CogTool can be useful, however, there is still a need to describe the task to be modeled, therefore, designers can only

obtain predictions of performance for tasks they have explicitly encoded. In complex legacy systems, there are many ways to accomplish a task, so modeling and analyzing all of these by hand becomes intractable.

1.1 Motivating Example

To provide a motivation for developing CogTool-Helper, we present the story of Sally, a UI designer and programmer for a company that makes office productivity applications. She is designing the interface for a new word processor and has developed a prototype of her application. She wants to compare it to the free open source word processor, OpenOffice Writer [39]. She wants to know if her application improves the efficiency of skilled users for common editing tasks. Sally knows about CogTool, a tool for obtaining human performance predictions for tasks as performed by a skilled user. Sally builds a CogTool design for both OpenOffice Writer and her new word processor, Magic-Edit, and comes up with a simple task to compare across the two word processors consisting of the following steps:

- ⋆ Type "`Chapter One`" into the document.
- ⋆ Select all the text.
- ⋆ Change font weight to Bold
- ⋆ Change text alignment to Centered.

There are three ways to perform each of last three steps of this task: using toolbar buttons, using menus, and using keyboard shortcuts. Sally has just added the toolbar functionality to provide quick access for common editing functions and would like to know if this addition improves user performance for Magic-Edit.

To get predictions for her task, Sally creates two CogTool designs; one for OpenOffice Writer and one for Magic-Edit. For each step in the task, she takes a screenshot

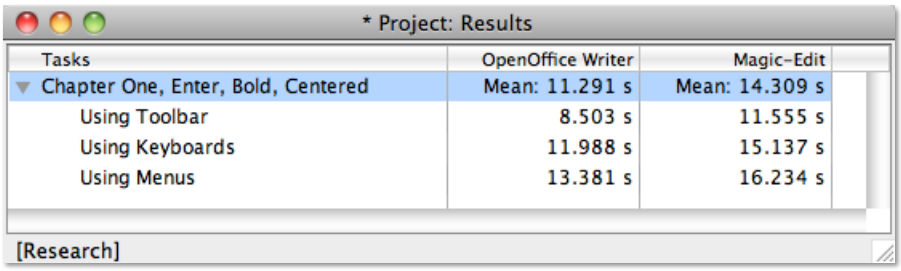


Figure 1.1: Sally's comparison of OpenOffice Writer and Magic-Edit

of the application. Then, she adds dynamic information. First, she highlights each of the widgets necessary for each method of her task (buttons, menus, and keyboard shortcuts) on the images, and draws the necessary transitions (representing the steps between the interface state represented by one image to another) between the images. Last, Sally demonstrates each of her methods on the CogTool model. She has to choose the image where the method begins and select the proper transitions to take to accomplish the task. She creates three methods for her task: `Using Toolbar`, `Using Menus`, and `Using Keyboards`. She repeats the entire process for both OpenOffice Writer and Magic-Edit. After method demonstration, Sally obtains predictions for her tasks (Figure 1.1) and sees that the `Using Toolbar` method is the most efficient way to complete this task in both applications, but performance for every method in Magic-Edit is almost 3 seconds slower. From this analysis, Sally is able to see that her new word processor design slowed down user efficiency, according to the CogTool predictions.

For an experienced user of CogTool, this process might take less than an hour. In this scenario, Sally has just started using CogTool, so it is likely that this process would take her several hours. She would also be likely to model the interface incorrectly because she is unfamiliar with CogTool and the way widgets are represented. If Sally wanted to evaluate several more complex tasks in addition to her simple one

and also compare Magic-Edit to Microsoft Word [37], Sally could easily spend an entire day's work on this analysis.

In addition, Sally wants to evaluate whether adding toolbars to her interface would be beneficial in all cases for this task and in all ways that this task could be performed. To do this, she would need to see that the majority of possible methods for this task that used toolbars were faster than those that used menus and keyboard shortcuts or a combination of both. Ideally, she would want to obtain human performance predictions for all possible methods that accomplish this task; those using only keyboards, only toolbar buttons, or only menus, or a combination of more than one, including all possible orderings of the actions in the task. There are more than seventy possible combinations and orderings for performing Sally's simple task. To check her assumption that the method using only toolbar buttons was the shortest possible method to perform this task, she would need to view predictions for all of the possible methods which would be nearly impossible for Sally to do without making any mistakes that might affect the CogTool results, and in a reasonable amount of time.

1.2 Leveraging Functional GUI Testing Tools

In testing of Graphical User Interfaces (GUIs), numerous tools and techniques have been developed for automated UI model extraction [31], test case generation [52], and test case replay [26]. Traditionally, system testers examined their applications that were to be tested, and then created manual use cases to exercise the important behavior [27, 50], but this approach has been shown to miss faults in the applications and to be time consuming to implement. In automated GUI testing, GUI widgets are represented as events in the form of a finite state machine [8] or a graph [28, 31]. Test case generation is performed by traversing nodes in the graph, and then the

generated test cases are replayed on the application. Using this automatic approach allows a much larger set of GUI events to be tested and a broader set of behaviors, and research has shown that automated GUI test case generation can improve fault detection [52].

In this thesis, we ask if it is possible to use GUI automation to help UI designers like Sally work more efficiently with predictive modeling tools. We introduce *CogTool-Helper*, an implementation of this idea. CogTool-Helper is a tool for automatically generating a human performance model for CogTool utilizing the GUITAR [30] GUI testing framework. CogTool-Helper can replay a test case on an application and convert this to a CogTool design. It can also analyze the existing frames and transitions of a CogTool design to uncover additional methods to perform a task beyond what the designer has specified. We believe such a tool will make the UI designer's life easier, especially when they want to compare new designs against an existing system. If Sally had used CogTool-Helper, she could have simply demonstrated her tasks while CogTool-Helper captured her actions. Then, she could simply click a button and watch CogTool-Helper do all of the work. At the end, she could import CogTool-Helper's results into CogTool and immediately obtain her predictions, saving her hours of work.

Every time a change in a GUI is made, the UI designer's analysis process must be repeated which is analogous to the process used in functional regression testing. We evaluate CogTool-Helper to see if it would be useful in a regression testing environment. We perform functional generation of GUI test cases to mimic a realistic task allowing us to explore all possible ways a user could accomplish a task on an interface. For UI designers like Sally, we believe such tools will ease the burden of their analysis process and allow concrete validation of the benefits and drawbacks of particular interface features such as toolbars . In Sally's case, test case generation to

generate a meaningful set of methods for a task will allow her to evaluate user performance for all combinations of menu, keyboard, and toolbar actions in all possible orderings for her task in much less time than it took her to build three methods and a design by hand in CogTool.

1.3 Research Contributions

This thesis presents CogTool-Helper, a tool to automatically generate predictive human performance models from legacy applications. CogTool-Helper is built upon existing tools for functional testing of graphical user interfaces. The user provides test cases as input and CogTool-Helper automatically converts these to CogTool designs. As a first step (presented in Chapter 3), we built a simple capture tool for the user to demonstrate a task and encode it as a test case. We also provide a mechanism to import test cases, and in Chapter 4, we extend CogTool-Helper to perform guided test case generation. A key feature of CogTool-Helper is an algorithm that traverses the designs and finds alternative ways for the user to perform the same task on the interface. We call these *inferred methods*. We evaluate the potential usefulness of CogTool-Helper in a study on human performance regression testing (Chapter 4), and show that we can provide useful information for validating the addition or removal of user interface features. We also evaluate the power of CogTool-Helper's inferred methods algorithm, and find that we decrease the time taken by CogTool-Helper to create designs and tasks by 50% while still delivering the same amount of information. This research makes the following contributions:

- We present CogTool-Helper, a tool for generating predictive human performance models from GUI applications using existing GUI test case replay tools.

- We present a method for describing a task in terms of a *test case*, generated by a testing tool or scripted by hand, and automatically transforming it into a CogTool *method*.

- We develop an algorithm to analyze the frames and transitions of a CogTool design to infer additional methods beyond what the designer has specified that accomplish the same task.

- We introduce a novel approach for regression testing of user performance that performs test case generation of realistic user tasks, and results in visualizations of user performance across a wide range of methods to accomplish the same task.

- We conduct a feasibility study to evaluate our approach showing that it can provide useful information about user performance, and that we can improve the efficiency of CogTool-Helper through sampling of test cases while retaining the same amount of information.

1.4 Overview of Thesis

This thesis presents background material on GUI testing and predictive human performance modeling in Chapter 2 with detailed descriptions of tools used in this work, GUITAR [30] and CogTool [18]. Chapter 3 presents our implementation of CogTool-Helper for generating predictive human performance models for CogTool and our algorithm for inferring additional methods from a CogTool design. Chapter 4 presents a method for generating meaningful test cases for a task utilizing the GUITAR framework and details a feasibility study evaluating this approach and the beneficial information it can provide in a regression testing scenario. We also evaluate the efficiency of our design construction in CogTool-Helper and show that it can

be improved through sampling of the test cases and relying on inferred methods to provide the rest of the information. Chapter 5 presents our conclusions and ideas for future work in this area.

Chapter 2

Background and Related Work

This chapter presents background on graphical user interfaces (GUIs), GUI testing, and the event-flow model for GUI testing. We also present background on predictive human performance modeling tools and techniques, including an introduction to CogTool, a tool for creating predictive human performance models of user interfaces or user interface designs.

2.1 GUI Testing

Graphical User Interfaces (GUIs) are the interface to a program. Most of today's software contains some sort of GUI. This is how most users interact with programs day-to-day on their computer. Users interact with objects in a GUI to obtain a response from the GUI. GUIs consist of objects such as buttons, menus, scroll bars, wizards, and windows and are hierarchical by definition. This can be seen in the way that events are grouped by dialog boxes, windows, and hierarchical menus.

We focus on only one class of GUIs in this thesis, those that are *event-driven*. When a user performs an action on a GUI, or the program dispatches a change in

the GUI automatically, and the state of the GUI changes in response, this is called an *event*. A GUI responds to an event with an *event handler* which is the piece of code in the program that effects a change in the state of the GUI in response to this event. Events are constantly changing the state of the GUI and therefore, their effects cannot be predicted. Event-driven GUIs have only a fixed set of events and each event has a deterministic outcome; each event causes a change in the behavior of the system. The important characteristics of these GUIs are their hierarchical structure, objects, properties and values, and response to events. This class of GUIs does not include web interfaces because of their synchronization and timing constraints, or non-deterministic GUIs because it is impossible to model the entire state of the GUI. With these types of GUIs, the effects of an event cannot be predicted. GUIs are automatically tested with *test cases*. These consist of a sequence of events in some order that can be replayed automatically on the GUI.

An important concept we utilize in CogTool-Helper is the current *state* of the GUI. The GUI state is defined as the set of all currently opened windows, their widgets, properties, and values. As the test case is replayed, each event changes the *state* of the GUI. The state is important when using an oracle verifier to compare the output from test cases. In GUI testing, test cases are run on a faulty and a clean version of an application (the oracle). The state is captured after each step and compared for fault detection. We describe the state here because capturing the current state of the GUI is an important concept that we utilize as part of CogTool-Helper's design generation process.

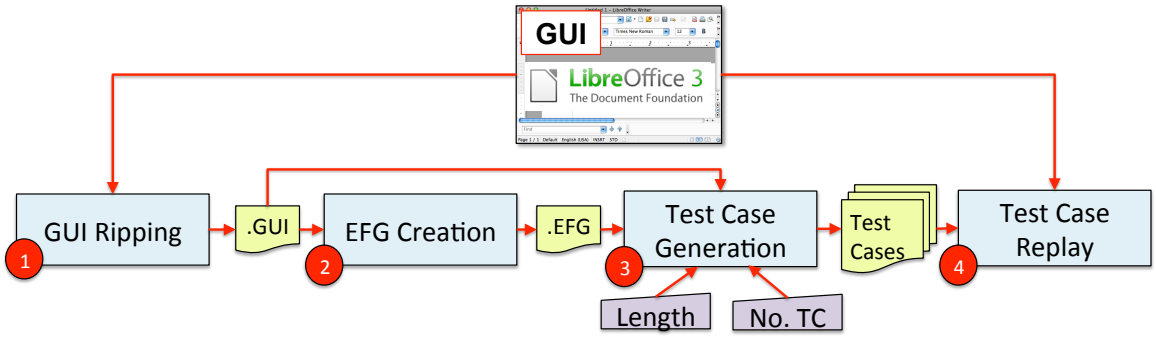


Figure 2.1: The GUITAR workflow

2.1.1 Model-Based GUI Testing

Techniques for model-based testing of GUIs are abundant in software engineering literature. These techniques have represented the GUI in several ways including finite state machines [8], graph models [31–34], and by visually describing the buttons and widgets [11]. GUI testing using pre and post-conditions [22] represents the GUI using an event-flow model but also tags this model with information about attributes that may effect the execution path (pre-conditions) and attributes that can change during the execution path (post-conditions).

The work focused on in this paper represents the GUI through graph models. The events that can take place on the GUI and their relationships are represented as an *event-flow graph* (EFG) representing all possible event sequences on a GUI. Test cases are created from this EFG by traversing a path and performing events along this path for a certain length. The GUITAR framework [30] puts this strategy into practice providing a way to create a model of the interface (EFG), a method to generate test cases, and tools to replay these test cases automatically on the interface.

2.1.2 The GUITAR Framework

GUITAR - A GUI Testing FrAmewoRk [30], from the University of Maryland, is a model-based system for automated GUI testing. The purpose of GUITAR is to automate the GUI testing process from start to finish including test case generation and replay. The GUITAR framework provides a fully automated end-to-end solution for testing event-driven GUI applications. The underlying strategy that GUITAR uses for testing is to reverse engineer the application's interface, obtaining a graphical model of the structure of the interface including the relationships between all GUI components. GUITAR then creates an event-flow-graph (EFG) for model-based testing. GUITAR can then generate test cases using various test generation algorithms and verify test results using a built-in oracle verifier.

The GUITAR framework supports various GUI application types including JFC, iPhone, OpenOffice, Android, Web, and SWT GUI toolkits. One of the benefits of GUITAR is that it is built with a plugin architecture and is an open source tool. Any developer can implement support for an unsupported GUI toolkit. GUITAR also allows for the implementation of various test case generation plugins that allow researchers and test engineers to implement and test their own algorithms for test case generation. The process of using GUITAR consists of four stages as depicted in Figure 2.1: **GUI Ripping** (Step 1), **EFG Creation** (Step 2), **Test Case Generation** (Step 3), and **Test Case Replay** (Step 4).

2.1.2.1 GUI Ripping

Looking at Figure 2.1, the first step (1) is GUI Ripping. The input to this process is the running application (GUI) and the output of this process is a GUI file (.GUI),

represented in an XML format, containing the GUI structure. The GUI file is then passed to the next phase, EFG Creation.

GUI *ripping* was first described in [31]. The ripping process extracts a model of the Graphical User Interface from the system. The ripper performs a depth-first traversal on the running application's interface opening all windows and widgets it encounters, gathering all information it finds on the GUI and its structure. The process of ripping is platform dependent, but the ripper does not need the source code of the application. The underlying mechanisms of the ripper for Java rely on Java Accessibility APIs [16] for automation while the ripper for Windows applications rely on Windows Accessibility APIs [35]. The use of accessibility APIs in GUI testing was first described in Grechanik, et. al. [15]. The ripping process consists of gathering a representation of the GUI in a graphical structure. This is referred to as a *GUI forest* in [31]. We refer to it as the GUI structure here.

The **GUI structure** represents the set of windows in the GUI, the hierarchical relationship between the windows in the GUI, and the structure of each window. In the GUI structure, the nodes are the windows of the GUI and the edges are the hierarchical relationship between the windows. Each of the windows of the GUI forms a hierarchy. The user first invokes the software, and any available windows at that point are *top-level windows*. All other windows of the GUI may be invoked from one of the top-level windows or one of their descendants. The relationship among GUI windows can be represented as directed acyclic graph (DAG) because multiple windows can sometimes invoke the same window. GUITAR reduces each DAG to a tree structure by copying nodes, and bases its algorithms for obtaining the GUI structure on tree traversals.

One factor influencing the GUI structure is the type of windows it contains [31]. GUIs contain two types of windows: *modal* and *modeless*. A modal window restricts

the user's interactions to widgets in that window while it is still open. Once it is closed, the user can interact with the other opened windows. A modeless window does not restrict the users' interactions to just that window.

In addition to containing the relationships between the windows, each node in the GUI structure also contains a window's widgets, their properties, and values. Each *widget* (e.g., buttons, text boxes, menus) within the window has a set of *properties* (e.g., label, font, background color) and *values* (e.g., Underline, Times New Roman, Red). The *state* of the GUI at a point in time is the set of all open windows, all widgets in each window, their current properties, and values. GUITAR can be instrumented to capture the current state of the GUI. GUI testing uses the state as an oracle to verify the behavior of a test case. In this work, we use the *state* of the GUI to construct CogTool designs representing multiple methods for a task (Section 3.3).

The resulting output from the ripping process is a GUI file containing the GUI structure. The GUITAR framework supports multiple customizations one can make to specify the areas of the GUI to extract and the amount of detail to be ripped. The tester can choose to filter out unwanted properties and implement filters to capture additional properties for specific widgets. At this point, however, the GUI is not useful for test case generation. Additional information must also be collected to model the *flow of events* of the GUI. This process is described in the next section, EFG Creation.

2.1.2.2 EFG Creation

The next step in the GUITAR process is EFG (Event-Flow Graph) Creation [28]. In Figure 2.1, this is Step 2. The input to this process is the GUI file (.GUI) created during Step 1. EFG Creation is platform independent. The output of EFG Creation is an EFG File (.EFG) in an XML format representing all of the event relationships.

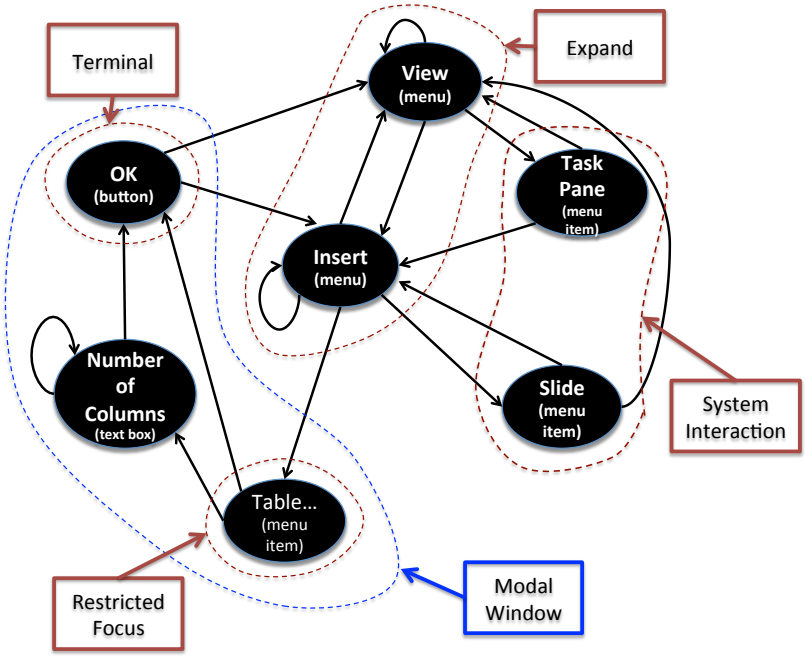


Figure 2.2: Sample Event-Flow Graph(EFG)

The EFG file can then be used to generate test cases for the application based on different algorithms.

The EFG Creation phase models the *flow of events* in the GUI. We provide the definition of an *event-flow graph* here.

Definition: An **Event-Flow Graph** (EFG) is a directed graph representing all of the possible event sequences that may be executed on a GUI. In an EFG, **nodes** represent the **events** that can take place on the GUI, and the **edges** represent the **relationships** between the events. If there is an edge from node x to node y, this means that the event represented by x can be immediately followed by the event represented by node y. This relationship is defined as a **follows** relationship.

A sample event-flow graph is shown in Figure 2.2. This EFG contains events that can perform three separate actions as executed in OpenOffice Impress (presentation

creation tool). These are inserting a new slide, showing or hiding the task pane, and inserting a new table. Each node is labeled with the name of the GUI component and each edge shows the *follows* relationship between the two components. In this EFG, the "View" menu has an edge going to "Task Pane". "Task Pane" is a child menu item of "View" so "Task Pane" can only directly follow "View". The "Insert" menu does not have an edge to "Task Pane" because it cannot immediately follow the "Insert" event. "View" must be performed first. The "Insert" event can, however, be followed by "Slide", which happens to be a child menu item of "Insert". A modal window can also be detected in this EFG. The "Table" menu item opens a modal window "Table" which has a "Number of columns" field and an "OK" button. The "OK" button can be reached from "Number of columns" but "OK" must be performed to close this modal window before any of the other events in the graph can be reached.

To enable the creation of the EFG, additional information must be captured during the GUI ripping process. This includes a classification of the GUI events that can be performed. Memon et. al [34] define 5 categories for classifying GUI events. These are: *restricted-focus*, *unrestricted-focus*, *terminal*, *expand*, and *system-interaction*. **Restricted-focus** events open a modal window. The user must interact with widgets in this window until it is closed. In Figure 2.2, "Table" is a restricted-focus event because it opens a modal window. **Unrestricted-focus** events open modeless windows. Other windows in the GUI can be interacted with while this window is opened. There are no unrestricted focus events in Figure 2.2. If the window opened by the "Table" event was non-modal, allowing outside actions in the GUI to be performed, the "Table" event would be an unrestricted-focus event. **Terminal** events close a window. The event "OK" in Figure 2.2 is an example because it closes the window opened by the "Table" event. **Expand** events open a menu or display a list of options. They have no interaction with the underlying software, but they

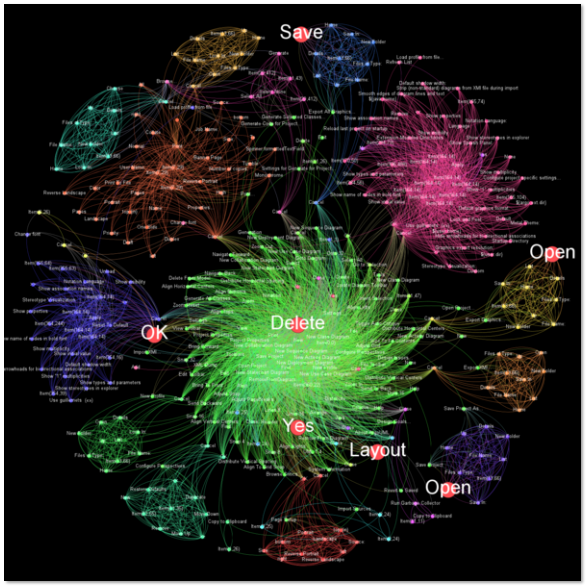


Figure 2.3: The ArgoUML EFG [30]

expand the set of GUI events available to the user (the "Insert" event opens a menu). **System-interaction** events interact with the software to perform an event, such as Copy or Paste (the "Slide" event inserts a new slide into the presentation).

GUITAR determines EFG relationships on the basis of these event types as well as the GUI structure. During EFG creation, GUITAR uses the information in the GUI structure to gather all of the possible GUI events associated with each component, and then determines how these components relate to each other. Each GUI widget may map to multiple EFG nodes. During ripping, GUITAR extracts all of the supported actions for each component (e.g., clicking, typing), and during EFG creation, an event is created for every supported action. The number and type of actions associated with each component is determined by the underlying implementation and GUI toolkit.

For very large applications, an EFG can become very complex. For example, Figure 2.3 [30], courtesy of the GUITAR website [1], is the EFG for the ArgoUML

[1] http://sourceforge.net/apps/mediawiki/guitar/index.php?title=The_EFG_galaxies

interface which has 423 nodes and 5,650 edges. The different colors represent the different interconnected areas of the GUI (windows and dialogs).

2.1.2.3 Test Case Generation

In Figure 2.1, Test Case Generation (Step 3) takes as input the EFG from EFG Creation (.EFG), the GUI structure file (.GUI), and two manual inputs, the length for the generated test cases, and the maximum number of test cases to generate. The output of this process is the set of generated test cases.

Test cases are made up of sequences of GUI events, and are generated by traversing the event-flow graph for the application and enumerating the events found. This allows a large number of test cases to be obtained quickly and efficiently. Test cases can be generated for specific lengths and according to different algorithms. The algorithms used depend on the areas of the application the tester wishes to cover and the quality with which they want to cover them. Memon et. al. [34] presents a set of *test coverage criteria* to guide the test case generation process. These are beyond the scope of this thesis, however.

This process of test case generation is platform independent. GUITAR provides two test case generators, Random Sequence Length, and Sequence Length. The Sequence Length generator traverses a given length sequence in the EFG to create the test case, and the Random Sequence Length generator traverses a random sequence of a given length in the graph. We utilize the Sequence Length generator for test case generation as described in Chapter 4. GUITAR is built to easily facilitate new test case generation implementations. Many of these experimental versions are provided through the GUITAR website[2]. After the test cases are generated, the test cases are automatically replayed on the interface during Test Case Replay.

[2]http://guitar.sourceforge.net

2.1.2.4 Test Case Replay

The final step of the GUITAR process is to replay the generated test cases on the GUI (Step 4 in Figure 2.1). The input to this process is the set of generated test cases created during Test Case Generation and the running application (GUI).

Test Case Replay is done using the *replayer*. The replayer takes as input a test suite or a test case to be replayed and automatically performs the test cases, step by step, on the interface. The replayer can be instrumented to calculate coverage for the application under test as well as extract state information to be compared with an oracle. The state extraction process captures the GUI state after each step in the test case which is aggregated into a state file output from the replayer. The *state* is similar to the GUI structure, but captures only the properties and values of the widgets on the currently opened windows.

The replayer is built to support plugins, allowing it to be instrumented with monitors to capture various aspects of the replay, such as the current state of the application. The replayer is platform specific. The underlying technology that the replayer depends on for automation is typically an Accessibility API for each platform but can differ based on the implementation and GUI toolkit.

2.2 Predictive Human Performance Modeling

Research into predictive human performance modeling has a 40 year history in HCI. This work has resulted in theories allowing reliable predictions of human performance, e.g.., the time it takes a skilled user to complete a task, as in CogTool, or the time it takes end users to learn the methods to accomplish tasks [17]. Very recently, researchers have attempted to predict the behavior of novice users, including task performance and errors made [48]. Adopters of these tools tell of value to their de-

velopment process from testing early UI design ideas, to evaluating proposed systems during procurement [17]. Modeling of such systems is not strictly limited to evaluating new design ideas but is often used to compare the performance of existing systems against proposed design ideas (e.g., five examples in Bellamy et al. [7], others in the work of Gray et al [14], Knight et al. [21] and Monkiewicz [36]).

When modeling performance on a user interface, user tasks must be described in the representation dictated by the human modeling framework. When modeling with GOMS (2.2.1) or KLM (2.2.2), tasks must be described through programming, as in GLEAN [20], or through demonstration, as in CogTool [18]. The work presented in this thesis allows tasks to be described in a third way; in the form of a *test case* generated by a test case generation tool or written by hand and then automatically transformed into a CogTool *method*.

2.2.1 GOMS

GOMS, introduced by Card, Moran, and Newell in 1983 [10], is a method for describing a task, and the user's knowledge of how to perform the task, in terms of Goals, Operators, Methods, and Selection rules. `Goals` are simply the goals of the user; what the user wants to accomplish using the software. Goals are often split up into several subgoals which are the smaller steps taken to reach the goal. An example of a goal is inserting a hyperlink. Subgoals might consist of opening an "Insert Hyperlink" dialog box, filling out the hyperlink address and text, and closing the dialog box. `Operators` are the actions the software allows the user to take, such as a button press or a menu selection. When inserting the hyperlink, operators could be "Click Insert menu" or "Type www.google.com". `Methods` are sequences of subgoals and operators that can accomplish the goal. `Selection rules` are rules users follow to decide which method

to use in a particular circumstance. For example, if a user is deleting a piece of text shorter than eight characters, they may hit the Delete key eight times. If the text is longer than eight characters, they would use the mouse to select the text and hit the Delete key. This would depend on the individual user.

A GOMS analysis can be useful for obtaining quantitative and qualitative evaluations of how users will interact with a system. There are several versions of GOMS, one of the most simple being the Keystroke-Level-Model (KLM) used by CogTool. The KLM uses only keystroke-level operators; no goals, methods, or selection rules.

2.2.2 The Keystroke-Level Model (KLM)

CogTool uses the Keystroke-Level Model (KLM) to make predictions of skilled user performance using a keyboard and mouse. Introduced in 1983, KLM [9], is a simplified version of a GOMS model for predicting how long it takes an expert user to perform a given task on a computer system. A task is broken up into a series of keystroke-level operators (e.g., keystroke, button press), and then these times are summed up to give a prediction. In KLM, a large task, e.g., editing a document, is broken up into a series of independent tasks called *unit tasks*. Card, Moran, and Newell [9] break each unit task into two parts: *acquisition* and *execution*. Acquisition consists of the user building a mental representation of the task and execution consists of the user executing a series of actions on the system to accomplish the task. The total time for the unit task is the sum of these two parts: $T_{task} = T_{acquire} + T_{execute}$. A unit task in the context of editing a large document would be centering the title of the document on the page. Execution time for a unit task is rarely more than 20 seconds. The KLM predicts only execution time and not acquisition time because acquisition time is more variable and largely depends on the individual context of each task and user.

In the KLM, each unit task has one or more *methods* that can be used to accomplish a task. These are different ways that the user knows to perform the task. Typically, expert users have one or more methods they know to accomplish a unit task. They can quickly choose which one to use at any instance, making their behavior predictable and routine, as opposed to a novice user, whose behavior is non-routine. Once the method is chosen, a KLM model can be constructed for the method.

A KLM model for a method is composed of a set of 6 primitive operators. These are: **K** (keystroking), **P** (pointing), **H** (homing), **D** (drawing), **M** (mental operator), **R** (system response). The **K** operator is a keystroke or a button press and its time is taken from an approximation of a standard typing rate. **P** is determined from *Fitt's Law* [46] which calculates the time required to point the mouse to a target on the display based on the distance from the target. **H** is the time required to move the hand between physical devices (e.g., mouse, keyboard). **D** is the time required to draw a straight line segment with the mouse. **M** is the mental preparation for a physical action, and **R** is the system response time for an action. Methods are represented as sequences of these operators where the total time for the method is:

$$T_{execute} = T_K + T_P + T_H + T_D + T_M + T_R$$

The sequence of KLM operators is determined by listing the physical operators (K, P, H, D, R) in sequence and then inserting mental operators (M) in the proper positions according to a set of five rules detailed by Card, Moran, and Newell [9].

Predicted times for each of these operators is determined from a number of experiments, theories, and heuristics. Refer to Card, Moran, and Newell [9] for a more detailed description of the KLM. CogTool forms the basis of its predictions for tasks

on this model, constructing a valid KLM for a task, and inserting mental operators automatically according to the rules.

2.2.3 ACT-Simple and ACT-R

ACT-R [4] (Adaptive Control of Thought - Rational) is a cognitive architecture for simulating and understanding human cognition. ACT-R forms the backbone of CogTool's human performance predictions. ACT-R is complex so we present only a brief description here. In ACT-R, models of human cognition are written in a script. Basic primitives and data types, determined to reflect the theoretical assumptions of human cognition, are expressed as a model which can be run in ACT-R to produce a quantitative performance prediction for the task. Running the model produces a step-by-step simulation of human behavior representing each individual cognitive action. Examples of these actions are memory encoding and retrieval, visual and auditory encoding, and mental imagery manipulation. Each step in the simulation is associated with a quantitative prediction for latency and accuracy. ACT-R is designed to simulate all actions taken by the human brain when performing a task.

The most basic component of the ACT-R model is a *production rule*. A production rule is a primitive statement representing a cognitive action; it is an *if-then* statement describing what the cognitive action will do if the condition is met. A cognitive task is achieved by stringing together a collection of production rules and applying them to working memory.

All ACT-R predictions have been determined by numerous psychology experiments collecting data from actual human behavior, brain imaging, and MRIs. In recent years, ACT-R has become increasingly popular in the field of human computer interaction; used to model information seeking behavior on the web [13], to predict

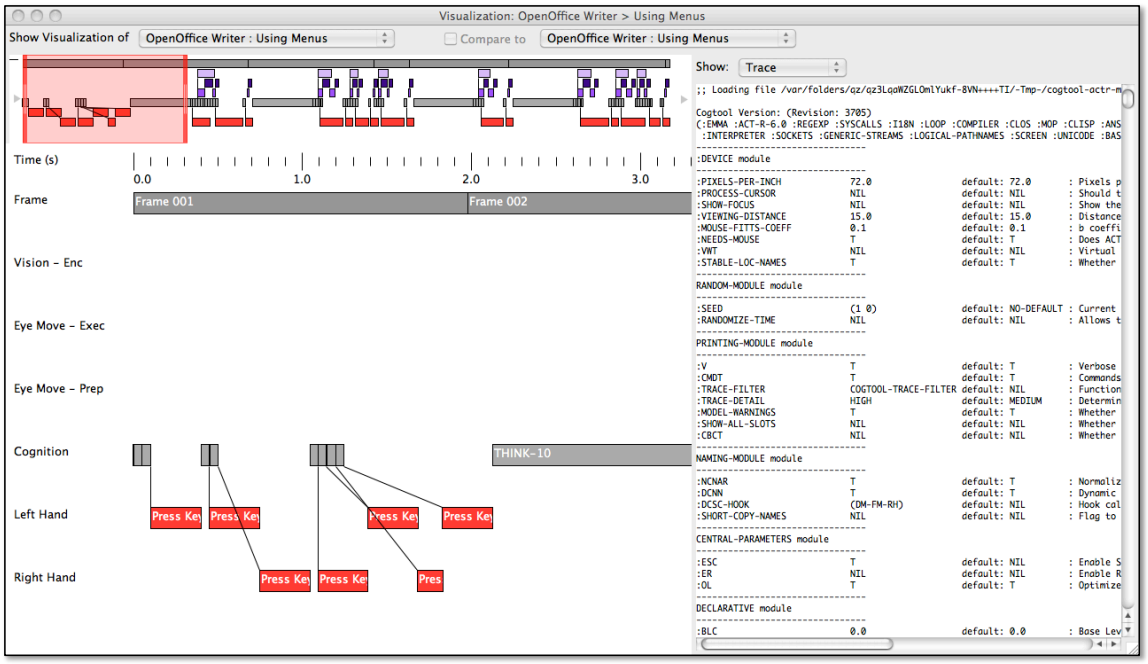


Figure 2.4: Timeline visualization of task in CogTool

the effects of cell phone dialing on driver performance [43], and in CogTool [18] to predict skilled execution time for user performance on tasks in user interfaces.

Instead of translating the KLM directly into the full detailed ACT-R model directly, CogTool uses a simplified higher-level version of ACT-R called ACT-Simple [44] to compile the KLM into a full ACT-R model. The ACT-Simple model consists of a set of high-level cognitive commands that are compiled into a set of lower level ACT-R production rules. These commands are *Move Hand*, *Move Mouse*, *Click Mouse*, *Press Mouse*, *Release Mouse*, *Press Key*, *Speak*, *Look At*, *Listen*, and *Think*. Each ACT-Simple command has a translation to the lower level ACT-R production rule. CogTool translates the sequence of KLM operators into a sequence of ACT-Simple commands, which it then compiles into the lower level ACT-R production rules. The purpose of ACT-Simple is to simplify the process of creating ACT-R models.

2.3 CogTool

CogTool [18] is a general purpose UI prototyping tool allowing UI designers to mock-up an interface and demonstrate tasks on that interface. In addition to more general purpose UI prototyping tools, it allows designers to obtain human performance predictions for designs across multiple tasks, and multiple designs. CogTool users create storyboards, an example of which is shown in Figure 2.5, representing tasks being performed on their interface. From these storyboards, CogTool creates a prediction of performance based on the ACT-R cognitive architecture and the Keystroke-Level-Model. Along with viewing performance predictions for a task or set of tasks on their interface, CogTool users can also view a timeline visualization (Figure 2.4) of the mental actions and motor movements CogTool has mapped to their task.

2.3.1 A CogTool Overview

The use of CogTool begins by a designer creating frames and transitions between those frames. Frames represent the different states that the application can be in as the task is performed (labeled in Figure 2.5). Transitions represent the actions a user takes on an interface; for example, "Left single-click on Bold button", to move from one state of the application to another. A storyboard such as this is called a design. Designs can contain one or more devices (mouse, keyboard, touchscreen, and microphone) that can be used to perform the task being modeled. Each CogTool frame is made up of widgets drawn on an image or a blank canvas which can be of several types (e.g., buttons, text boxes, menus). CogTool has several types of widgets available (Figure 2.6). Widgets are the highlighted components that the CogTool user manually highlights on each frame. Transitions are the arrows connecting the frames.

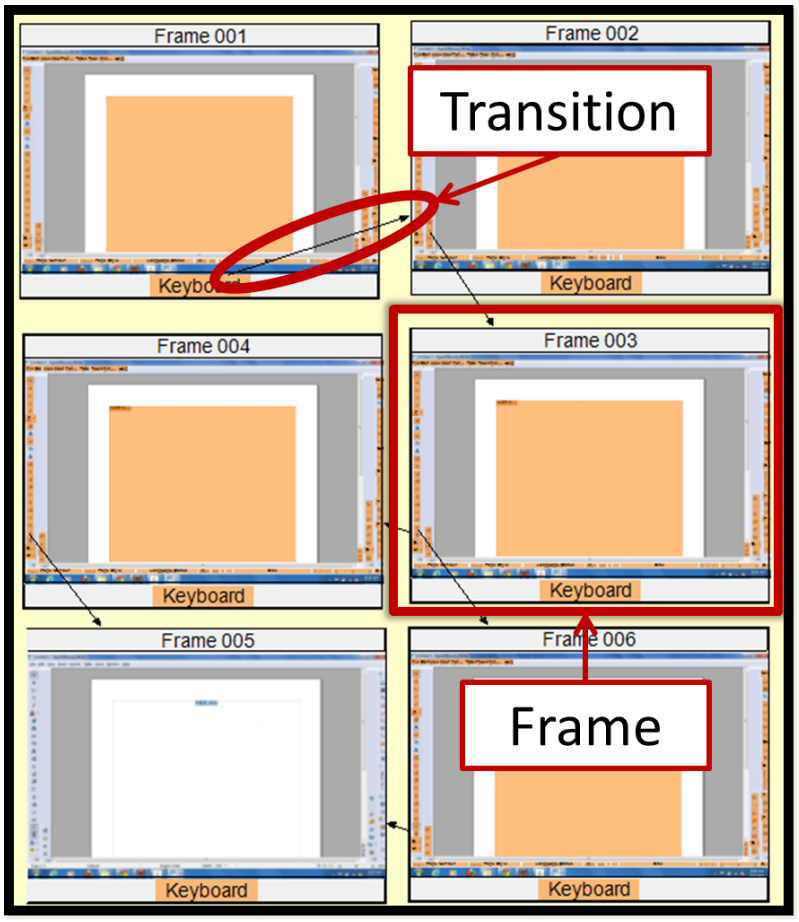


Figure 2.5: Sample CogTool design for OpenOffice Writer

Actions through widgets, or through audio and touch input, trigger a transition to the next frame.

After the designer has constructed a storyboard, it can then be used to build a human performance model. In CogTool, this is implemented through the KLM. To build this model, the designer must demonstrate a task on their design. This process of demonstration starts with the designer selecting a start frame for the task and then selecting the appropriate transition to take to the next frame. As the designer demonstrates the method, CogTool collects the operators, and inserts additional mental operators, for the KLM model. Once the script is built, a CogTool

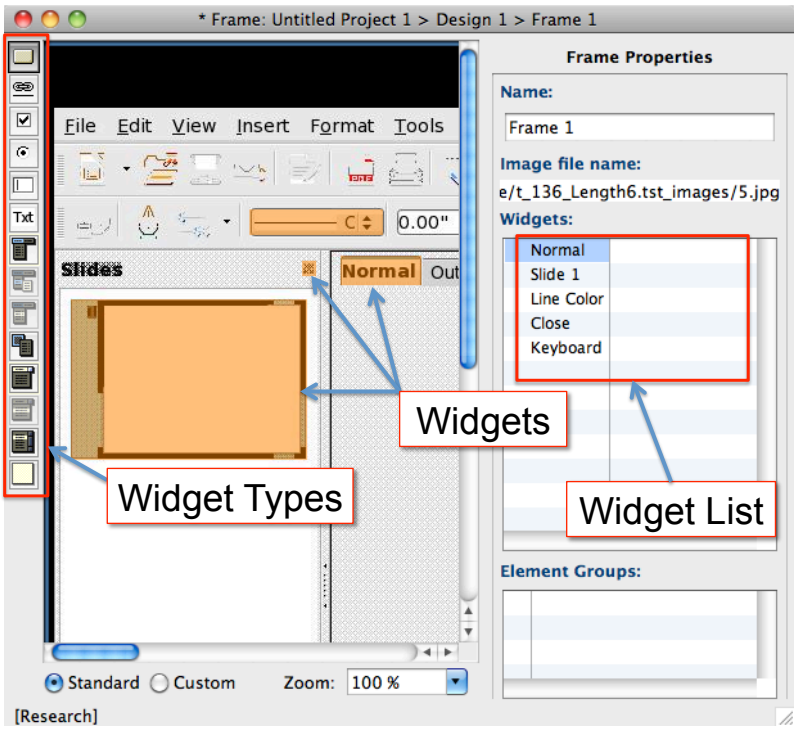


Figure 2.6: Constructing a CogTool frame

user can look at the steps of the task in the script window (Figure 2.7). After the model is built, the designer clicks a "Compute" button, and a computation is performed to obtain a human performance prediction for the task (the time it would take a skilled user to perform the task). CogTool currently supports predictions only for skilled users.

With CogTool, a designer can compare predictions across methods and tasks and explore the efficiency of different UI designs across these tasks. Figure 2.8 shows how two designs, OpenOffice Writer [39] and Microsoft Word [37], could be compared for the task "Insert Hyperlink" with the methods `Using Toolbars` and `Using Menus`.

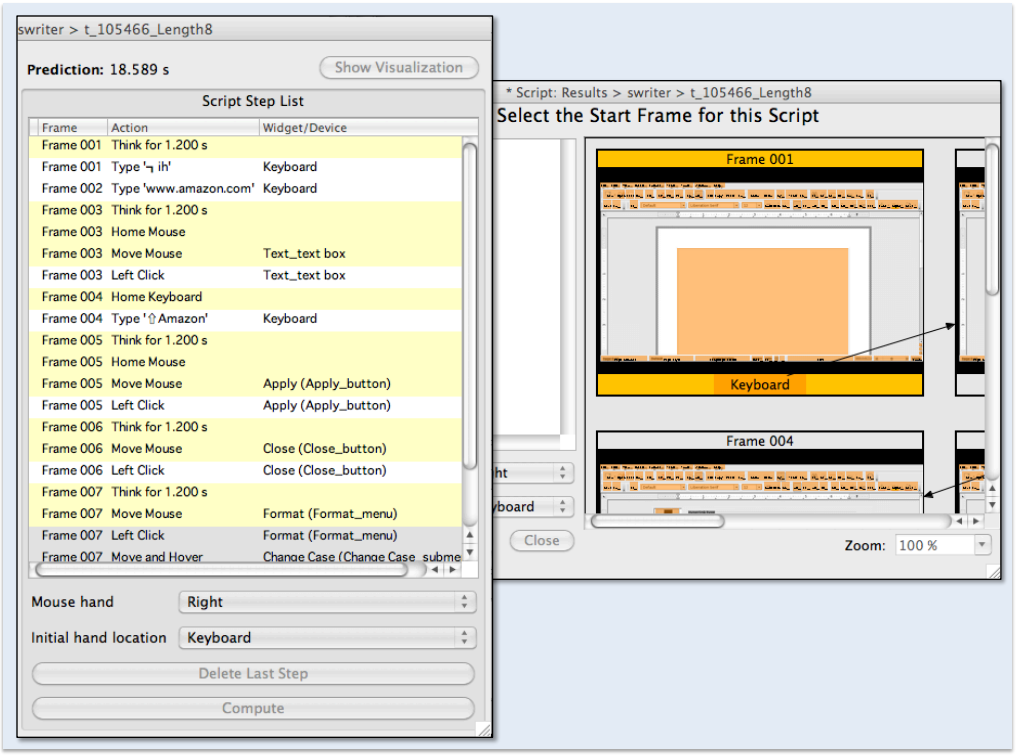


Figure 2.7: CogTool script window

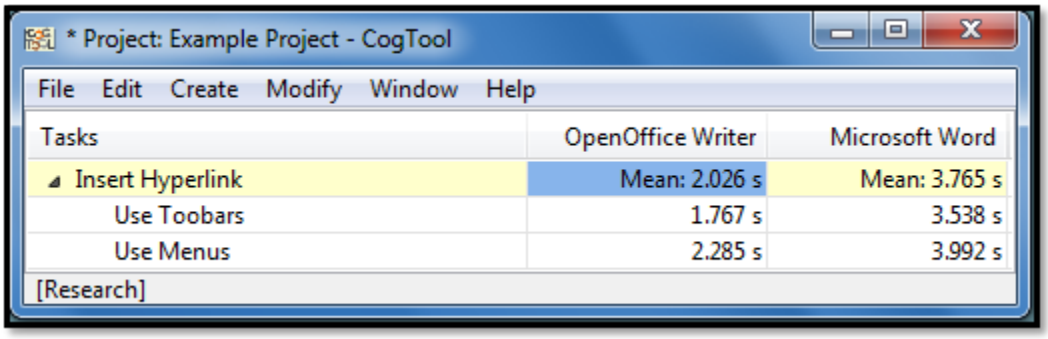


Figure 2.8: CogTool comparison of OpenOffice Writer and Microsoft Word

2.3.2 CogTool's Predictive Model

CogTool goes through several processes to obtain its performance predictions (Figure 2.9). First, the UI designer demonstrates the task on the model. As they are doing this, CogTool constructs the KLM and inserts the KLM mental operators.

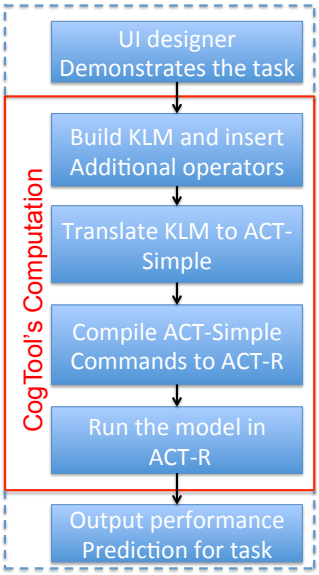


Figure 2.9: CogTool's prediction process

Next, it translates the KLM script into a series of ACT-Simple commands. Then, these are compiled into their lower level ACT-R production rules. Final, the model is run in ACT-R, and ACT-R outputs the performance prediction which CogTool then displays to the user.

2.3.3 CogTool's Uses

CogTool was first introduced in John et al. [18] in 2004. Since then, it has been used in numerous domains to predict skilled-user performance. Bellamy et al. [7] demonstrated the successful use of CogTool in an industrial context in four scenarios: contract compliance, communication within a product team and between a product team and its customers, assigning personnel to customer complaints, and quantitatively assessing design ideas. Agarwal et al. [2] used CogTool to develop a measure of intuitiveness and better understand user's experience of their company's web applications. Recent work has expanded CogTool's domain to mobile phones [19], touch-

screens [1], and PDAs [25]. The reach of CogTool has expanded outside the area of predicting human performance to model energy consumption [24], where system activities consuming energy are correlated with user actions in the KLM.

Chapter 3

CogTool-Helper

With the advent of CogTool and predictive human performance modeling techniques, there is a need for modeling legacy systems. Models of legacy systems can be used to compare these applications to their competitors, or propose new design ideas for the existing version of an application. CogTool-Helper fills this HCI need by leveraging existing work in the field of automatic GUI testing. In CogTool-Helper, we utilize the GUITAR framework to automate the creation of predictive human performance models for CogTool. CogTool-Helper uses information that is freely available from the GUITAR test case replayer and its underlying framework to create a CogTool storyboard and a set of methods for tasks from which a UI designer can immediately obtain performance predictions for execution times by skilled users.

This chapter presents the details of CogTool-Helper's operation and implementation details, and then describes how we can use a storyboard created by CogTool-Helper to infer additional methods for a task beyond what the UI designer has explicitly specified. Some of the material presented in this chapter can also be found in [47].

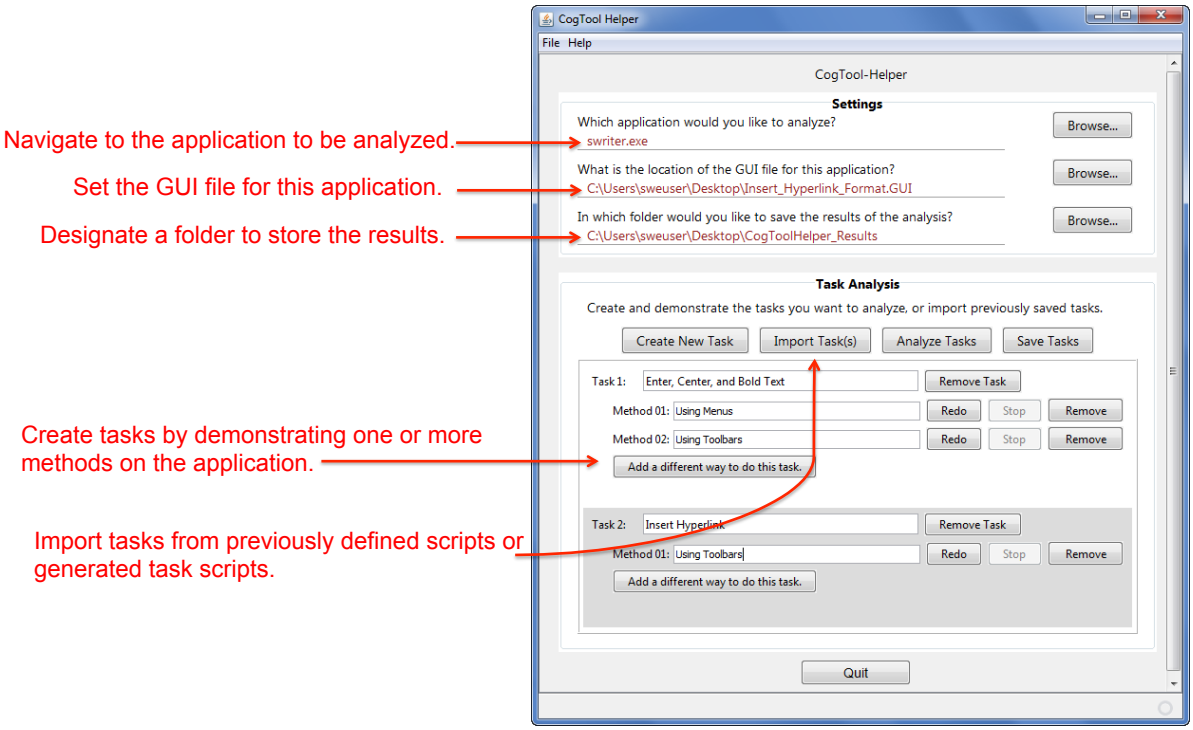


Figure 3.1: CogTool-Helper interface

3.1 Overview

The input to CogTool-Helper is a test case or set of test cases. In CogTool-Helper, we map a GUI *test case* to a *method* in CogTool. In CogTool, a *task* is a concrete goal a user is trying to accomplish on the interface. A *method* consists of the specific steps the user takes to accomplish the goal. There can be many methods for performing a single task. A GUI test case corresponds to a method in CogTool, and a set of GUI test cases that all perform the same set of functional changes in the system is a *task* in CogTool.

CogTool-Helper is a standalone Java application (Figure 3.1) built on top of the existing test case replayer, GUITAR [30]. CogTool-Helper currently works on Java and OpenOffice applications only, but GUITAR is also able to support SWT, Android, iPhone, and Web applications so support for these application types will be added as

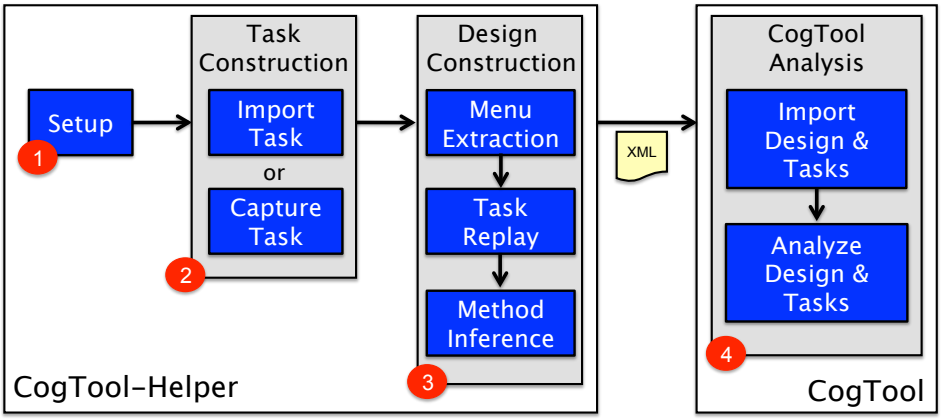


Figure 3.2: CogTool-Helper's process and relationship to CogTool

future work. Access to the source code is not required to run CogTool-Helper because GUITAR uses Accessibility APIs [15] to connect to the UI for automation and replay.

Using CogTool-Helper consists of two main phases, Task Construction and Design Construction as illustrated in Figure 3.2 and presented in detail below. The output of this process is an XML file representing a complete CogTool model for the set of constructed tasks that can then be imported into CogTool for design and task analysis.

3.2 Operation

The operation of CogTool-Helper consists of three phases. The first is the Setup phase where the designer selects the application they want to analyze and chooses the location to store the results of their analysis, described in Section 3.2.1. The second phase, Task Construction, described in Section 3.2.2, consists of the designer constructing the tasks and methods that they are going to analyze on their application. The third phase, Design Construction, involves no interaction from the user. During this phase, CogTool-Helper creates the CogTool designs and tasks for their selected

application and demonstrated tasks. Design Construction is described in more detail in Section 3.3.

3.2.1 Setup

When CogTool-Helper is launched, the designer selects a legacy application to be analyzed. In Figure 3.1, the OpenOffice text editor (swriter.exe) has been selected. Next, if the designer is importing a set of generated test cases, they will need to set the location for the GUI file for those test cases. Otherwise, this setting is not required. Lastly, the designer must choose a location to store the results of their analysis (C:\. . . \CogToolHelperResults in Figure 3.1). This is where an XML project file will be stored that can be imported into CogTool. Once these parameters have been set, the process of Task Construction can begin.

3.2.2 Task Construction

In the Task Construction phase, the UI designer defines one or more tasks and creates one or more methods to achieve each task. As shown in Figure 3.2, there are two different ways to achieve this goal, Import Task, where the designer loads in previously defined tasks and methods, or Capture Task, where the designer demonstrates methods on the application being analyzed, and CogTool-Helper captures their actions. Each method is stored in CogTool Helper as a GUI test case in the GUITAR format. This allows for scripting and automatic generation of methods and makes for easy integration with the test case replayer.

As an example, consider using OpenOffice Writer to enter text, center it on the page, and make it bold (Enter, Center, and Bold Text). A UI designer has entered two methods for doing this task in CogTool-Helper (Figure 3.1), one method using all

menu actions and the other using all toolbar button actions. These demonstrations create two test cases in the GUITAR representation. Demonstrated methods are represented as a test case with a series of steps. Each *Step* is composed of an `EventID`, `Window`, `Action`, `ReachingStep`, and `Parameter`. The `EventID` and `Window` properties help GUITAR identify the widget on the screen. The `Action` tells GUITAR what action to perform for that step. This will be one of several action types that GUITAR supports including *Type*, *Click* (left), *Select From List*, *Keyboard Access*, and *Keyboard Shortcut*, and *Set Value*, each corresponding to a GUITAR action handler. We have implemented support for keyboard shortcuts and keyboard access as an addition to GUITAR. The next property is `ReachingStep`. This property is set to true for each step added to a test case to allow the starting event in a test case to be reached. For captured test cases, we set this property to false. The last attribute included in the test case step is `Parameter` and is required only for the *Typing* action. We have implemented various parameters that allow the test case to perform more advanced text editing functions not supported by GUITAR. These include the ability to type a piece of text, move the cursor to a specified location in the text, select a portion of the text, and unselect the text. The captured test case step for clicking the Select All button in OpenOffice Writer is represented as follows.

```
<Step>
   <EventID>Select All_62<EventID>
   <Window>Untitled 1 - OpenOffice.org Writer</Window>
   <Action>edu.umd.cs.guitar.event.OOActionHandler</Action>
   <ReachingStep>false</ReachingStep>
   <Parameter></Parameter>
</Step>
```

This step would be slightly different for Java interfaces because GUITAR needs slightly different information for EventID and Window to identify the widget. The

Action (edu.umd.cs.guitar.event.OOActionHandler) corresponds to the GUITAR action handler for the 'Click' action.

CogTool-Helper also includes support for replaying generated test cases. These are automatically created by GUITAR so they have a slightly different format. GUITAR assigns each widget a unique ID during the ripping process so it does not need the Window property to identify the widget. It also does not need the Action property because an action is associated with an EventID. A step in a test case is a reaching step (`ReachingStep`) if it was added to the test case only to allow the starting event in the test case to be reached which can be true or false for automatically generated test cases. A Parameter is allowed but not required. The automatically generated step for clicking on the Select All button would look like the following.

```
<Step>
   <EventID>e111378976<EventID>
   <ReachingStep>false</ReachingStep>
</Step>
```

Different methods to achieve the same effect result in different GUITAR test cases. For instance, when using the menus instead of the toolbars, selecting the all text takes two steps, an event to click on the Edit menu and an event to click on the Select All menu item.

```
<Step>
   <EventID>Edit_34</EventID>
   <Window>Untitled 2 - OpenOffice.org Writer_ 49</Window>
   <Action>edu.umd.cs.guitar.event.OOActionHandler</Action>
   <ReachingStep>false</ReachingStep>
</Step>
<Step>
   <EventID>Select All_44<\EventID>
   <Window>Untitled 2 - OpenOffice.org Writer_49</Window>
```

```
    <Action>edu.umd.cs.guitar.event.OOActionHandler</Action>
    <ReachingStep>false</ReachingStep>
</Step>
```

Tasks can be scripted by hand or generated using an automated test case generator (as we will show in Chapter 4), and then loaded into CogTool-Helper with the Import Task button shown in Figure 3.1, but CogTool-Helper also allows designers to demonstrate methods using the capture task feature. To capture a new method for a task, the designer provides names for the task and method in the appropriate text fields (Figure 3.1). Next, the designer clicks "Start", and CogTool-Helper launches the legacy application. The designer then demonstrates the task on the legacy application. This consists of performing each step of the task by hand on the interface. CogTool-Helper captures each steps and converts it to the test case format described above (for captured test cases). The designer clicks "Stop" when s/he has completed the task; recording stops, and the method is saved as a test case.

In Figure 3.1, we see CogTool-Helper with two tasks defined. The first task, "Enter, Center and Bold Text", has two methods: "Use Menus" and "Use Toolbar". The second task, "Insert Hyperlink", has only one method, "Use Toolbar". The first task has been imported from a file, but is indistinguishable from the other task that was captured by the designer. Once the designer has created and saved all of the tasks to be analyzed by CogTool, s/he will click the "Start Analysis" button which begins the second phase of CogTool-Helper, Design Construction.

3.3 Design Construction

The goal of the design construction phase is to generate all the information needed by CogTool to model a UI design and tasks performed on it and represent this informa-

tion in XML so it can be imported into CogTool. There are four key processes that contribute to this phase of CogTool-Helper: Menu Extraction, Design Initialization, Task Replay, and Method Inference.

3.3.1 Menu Extraction

CogTool-Helper captures simple widgets (e.g., buttons, text boxes) as they appear during Task Replay, but for menus and pull-down lists, it extracts them during the menu extraction process. This is analogous to GUI ripping, and is done in order to construct the CogTool widgets for the menus. Menu Extraction is done only once before any of the methods are replayed. CogTool-Helper systematically opens all menus, pull-down lists, dialog boxes, and any other elements that are not initially visible within the root window of the application, via a depth first traversal. This must be done to construct these widgets because the child elements of these widgets (menu items, etc.) are not available to the GUITAR replayer when the menu or list is not expanded. CogTool-Helper's menu extraction records the size, position, label and type of these widgets in CogTool XML format. This set of widgets will be used in the next process, Task Replay, so appropriate widgets can be added to the frames in the CogTool design. This process also assumes that these objects will not change location or properties throughout method replay. For this reason, CogTool-Helper currently supports only interfaces with static menus. To support dynamic menus that change as a method is being performed, menu extraction would have to be repeated before every step of every method. This might be intrusive to the user as well as slow down design construction. Future versions of CogTool-Helper may allow the designer to change the process for menu extraction to support dynamic menus.

Menu extraction can take a considerable amount of time for complex interfaces, even with it opening hierarchical menus as fast as possible, so it is shown to the designer as feedback that CogTool-Helper is working. Once menu extraction is done, CogTool-Helper proceeds to the Design Initialization stage.

3.3.2 Design Initialization

Before replaying any test cases, CogTool-Helper must create the initial XML for the design. Each CogTool design must have at least one device. Possible devices are mouse, keyboard, touchscreen, and microphone. CogTool-Helper supports only mouse and keyboard devices currently, so these two devices are included by default for every design. In CogTool designs, the design name is the name of the application being analyzed. The initial design for OpenOffice Writer is encoded as follows.

```
<design name="OpenOffice.org␣Writer">
   <device>mouse</device>
   <device>keyboard</device>
</design>
```

3.3.3 Task Replay

During task replay, a CogTool design that supports all tasks specified by the designer during task construction is created incrementally (Figure 3.3). Each method of each task is automatically performed on the interface using the GUITAR replayer which treats each method as a test case and performs each step in turn on the UI. We have modified GUITAR to capture the UI information that CogTool needs and translate the test case into a CogTool design. As with menu extraction, the task replay process takes time and it is shown to the designer. But instead of opening all menus and lists, task replay performs just those actions recorded in the test case for accomplishing

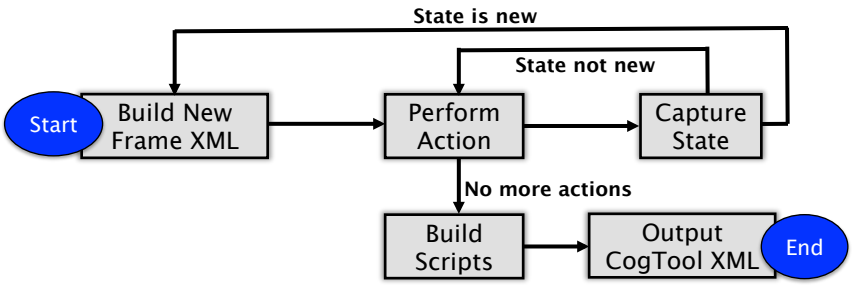


Figure 3.3: Task Replay workflow

this task. Between each replayed test case, CogTool-Helper closes the application, reinitializes all user preferences and data for the application, and restarts the application fresh so that each test case is replicated in the same environment, and the unseen effects one test case might have on the application do not interfere with the behavior of any other test cases.

The task replay process consists of five phases: Build New Frame XML, Perform Action, Capture State, Build Scripts, and Output CogTool XML. These are described in the following sections. The workflow of Task Replay is shown in Figure 3.3.

3.3.3.1 Build New Frame XML

If this is the first method being analyzed, CogTool-Helper starts with Build New Frame XML, which constructs the XML for the initial CogTool frame. Build New Frame XML begins by capturing a full screen image of the current desktop using the java.awt.Robot class. The image data is placed in the background slot of CogTools XML for this frame (*backgroundImageData*). Because all the background images are full screen, and the background image sets the size of a frame in CogTool, all frames are the same size and scale; keeping all frames at the same scale is important for CogTools human performance model to make accurate predictions of the duration of mouse movements. Each frame must have a name and we create these incrementally

as a new frame is built and name them Frame 001, Frame 002, etc. In each frame, we will create CogTool widgets for all widgets found on the current screen (within the application). Most OpenOffice modules have four to five hundred widgets that CogTool-Helper collects. The frame is encoded as follows.

```
<frame name="Frame 002">
   <backgroundImageData>.....</backgroundImageData>
   <topLeftOrigin y="11" x="11"/>
   <widget>....</widget>
   <widget>....</widget>
    ....
</frame>
```

CogTool-Helper then determines which windows are visible at this point in the task from the replayer. CogTool-Helper is also able to tell whether a window is modal or non-modal. If a window is modal, CogTool-Helper constructs only the widgets in the active modal window on this frame because a user, and therefore the CogTool model, can interact only with the modal window at this point in the task. If the window is modeless, CogTool-Helper constructs widgets from all open windows because a user (and the CogTool model) could interact with any of them.

To get the widgets in a frame, CogTool-Helper traverses the accessibility trees provided by the application for each window, collecting every object corresponding to a CogTool widget (e.g., buttons, links, text boxes). Each accessibility object is provided to CogTool-Helper by GUITAR which provides functions for extracting certain properties. We use a subset of these properties in CogTool-Helper. In addition, when GUITAR does not provide the properties we need, we extract them directly from the accessibility object. Each object has an internal *role*. GUITAR calls this the `Class` of the widget. The `Class` of a widget corresponds to the type of widget it is. For example, in the UNO API, widgets are assigned classes such as PUSH_BUTTON for a

button that does not toggle after it is clicked, and MENU for a top-level menu widget. For Java JFC widgets, examples are *javax.swing.JToggleButton*, which represents a button that toggles, and *javax.swing.JCheckBox*, which represents a checkbox widget. For each new GUI framework, such as the UNO and JFC, we need to define a mapping between each widget Class and its corresponding CogTool widget type. We have defined mappings for only these two frameworks so far.

In addition to determining the CogTool widget type for an accessibility object, we need to tell CogTool the `size` and `position` of the widget. For each widget, CogTool-Helper extracts the position of its `upper left corner`, its `height` and its `width`. These correspond to the GUITAR properties `X`, `Y`, and `Size`. CogTool-Helper translates these properties into the `Extent` of the widget in CogTool XML. This will tell CogTool where to highlight the widget on the image in the current frame.

For each widget, we also create a `label`. This gives the widget a textual label in CogTool. The label corresponds to the `Title` property in GUITAR. This is the textual label associated with the widget. If the widget does not have a textual label, its Title corresponds to the tooltip text found when hovering over the widget (as in the UNO interfaces, which have toolbar buttons with an icon instead of a textual label).

CogTool also requires a `name` for each widget. This also corresponds to the `Title` property in GUITAR. In the case of non-unique titles, we add an `ID` onto the end of the name so each widget on a frame can be uniquely identified. CogTool does not allow widget names to be duplicated on any frame.

For some CogTool widget types, additional information must also be captured. Some of this more fine-grained information is not provided by the GUITAR framework, so we extract it directly from the accessibility object. For example, CogTool has a property `Is Toggled` that can be set for a button widget if it is a toggle button

and is currently pressed down. We can extract the state of a toggle button from the accessibility object from which we can tell whether the button is currently toggled.

Once CogTool-Helper has gathered all of the necessary widget properties, it creates a new widget in the CogTool widget format, adding it to the current frame. A example frame created by CogTool-Helper for OpenOffice Writer, with its first widget (the Select All toolbar button), is encoded as follows.

```
<frame name="Frame 002">
   <backgroundImageData>.....</backgroundImageData>
   <topLeftOrigin y="11" x="11"/>
   <widget w-is-selected="false" w-is-standard="true"
      name="SelectAll_button_1" shape="rectangle"
      w-is-toggleable="false" type="button">
      <displayLabel>Select All</displayLabel>
      <extent height="27" y="82" width="25" x="1020"></extent>
   </widget>
</frame>
```

3.3.3.2 Perform Action

After the frame has been built, CogTool-Helper proceeds to the next phase of Task Replay, Perform Action. Perform Action looks at the action at this point in the test case and begins to create the CogTool XML representation for a transition. This step in the test case says which widget is the source of the transition and what type of transition it is (e.g, mouse click, keystrokes on the keyboard), but not what frame it will transition to. Perform Action performs the action using the associated GUITAR action handler. The transition for clicking on the Select All button is created as below. We do not yet know what the destination frame will be.

```
<widget>
   <transition durationInSecs="0.0"
```

```
        destinationFrameName="....">
        <action>
           <mouseAction button="left" action="downUp">
           </mouseAction>
        </action>
     </transition>
  </widget>
```

The action for clicking on the Select All button above in the test case was the "Click" action. CogTool-Helper uses the action handler found in the test case to map to the correct CogTool transition. Each action handler maps a specific CogTool transition type. Available actions are *Left-Click*, *Type*, *Set Value*, *Select From List*, *Keyboard Shortcut*, and *Keyboard Access*, each of which maps to a specific GUITAR action handler. The Type action takes additional parameters. These are *TextInsert*, *TextReplace*, *Select*, *Unselect*, *Cursor*. Some of these can take additional parameters based on the text to be typed or the location to move the cursor. A guide to each of these actions and their corresponding representation as CogTool transitions is given in Appendix A.2.

3.3.3.3 Capture State

Perform Action creates a new current state of the application which is passed to the Capture State process. The purpose of capturing the state is to determine whether CogTool-Helper needs to build a new frame for this state, or whether it should link to an existing frame.

The state of the application consists of all information that can be obtained through all widgets on the interface. It consists of attributes such as text in the document, the current font, the current font size, and the current selection state of a toggle button. GUITAR captures state information for comparison with a pre-defined

oracle when used in automated GUI testing. We have modified GUITAR to capture some extra details for our purpose. In determining which properties to capture for the current state, we made an examination of all the available properties through GUITAR and selected just those that we thought the UI designer would be able to see on the interface and use to differentiate each frame when they are constructing it by hand. This does not need to be as detailed as a GUI oracle. In addition, extracting too many properties may slow down the process of creating CogTool designs.

Next, CogTool-Helper must decide which frame to transition to. CogTool-Helper keeps a list of all states that have been encountered while building the design, each of which is linked to a particular frame. If the current state does not match a state in the list, CogTool-Helper places the new current state in the list, maps it to an empty frame, and sets the target of the transition to the empty frame. If it is in the list, CogTool-Helper sets the target of the transition Perform Action has just created to the frame associated with the current state. The transition associated with the Select All button in Frame 002 shown above, is now encoded with the destination frame name set to the name of the frame that matches, or the name of the new empty frame.

Next, if the current state was not new, CogTool-Helper repeats the Perform Action phase for the next step in the test case. If the state was new, CogTool-Helper does not proceed to the next step in the test case but instead goes back to Build New Frame XML to fill in the empty frame.

3.3.3.4 Build Scripts

Once there are no steps left in the test case, CogTool-Helper has finished representing this method in CogTool XML and returns the set of states and frames that have been defined so far. These will be the input to the next method if there are any left to

process. CogTool-Helper proceeds in the same way for every method of every task, keeping track of which frames and transitions are needed by each task (used by the method inference process described in Section 3.4). Once all tasks are complete, the CogTool XML contains all the information for a complete CogTool design storyboard, with all frames, widgets and transitions created by each of the replayed test cases.

The last part of the Task Replay process (Figure 3.3) is building CogTool scripts, i.e., representations of the demonstrated steps in each method in CogTool XML form, for each of the replayed test cases. Scripts include the mouse and keyboard actions assembled by CogTool-Helper, to which CogTool will add psychologically valid un-demonstrated steps, like eye movements and thinking time, when it builds the ACT-R cognitive model. In CogTool, a script represents the steps that would be taken by the user in performing a specific method of a task. Scripts can also be created by demonstration in CogTool as well (described in Section 2.3.1). CogTool provides an XML format to encode scripts, so for each test case, we encode a CogTool *script* so that a UI designer does not have to manually demonstrate the method. CogTool immediately computes these scripts on import so designers can see predictions of skilled user execution time for their tasks immediately upon import.

For each user action in a method (step in the test case), CogTool-Helper creates a demonstration step in the script with the widget on which the action was performed and the type of action (e.g., mouse, keyboard). In the following step, the left mouse button is clicked on the Select All button. For each action that was replayed in the test case, a similar step is created. The type of the demonstration step (mouse or keyboard) depends on the type of transition that was created when executing the Perform Action step on the widget. Perform Action creates a transition and associates it with the specified widget. When CogTool-Helper creates the script for

this method, it translates each of the transitions created during Task Replay of this test case directly to demonstration steps for this method.

```
<demonstrationStep>
   <actionStep targetWidgetName="SelectAll_button_1">
      <mouseAction button="left" action="downUp">
      </mouseAction>
   </actionStep>
</demonstrationStep>
```

After script creation, CogTool-Helper computes the inferred methods and creates scripts for them. We describe how this is done in Section 3.4. CogTool-Helper then outputs the newly created CogTool project in the location that the user specified during the Setup step. The UI designer can now import this as a project into CogTool and view their predictions.

3.3.4 Import Design & Tasks

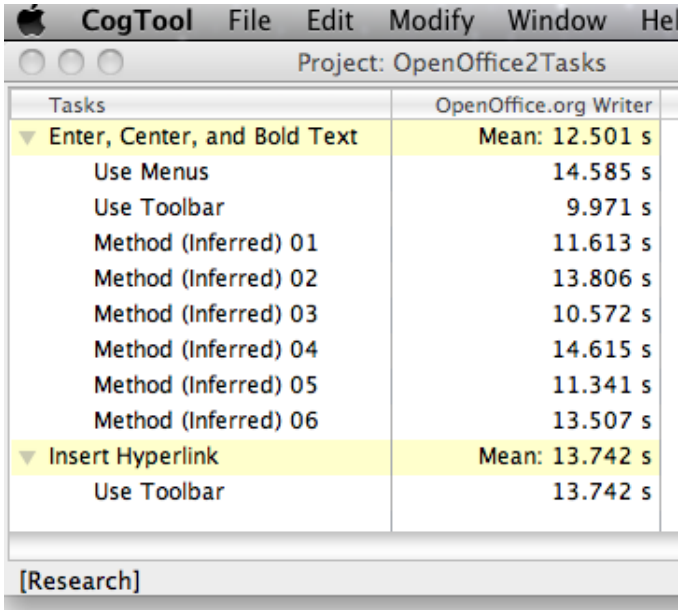


Figure 3.4: The imported design, tasks and predictions

Once CogTool-Helper is finished, the designer can launch CogTool in either MacOS or Windows and import their designs and tasks from the XML file. We have added an option to CogTool that will automatically construct and run the ACT-R model and calculate predictions for each task upon loading. This way, the designer can open the XML file and immediately compare each method of each task for performance.

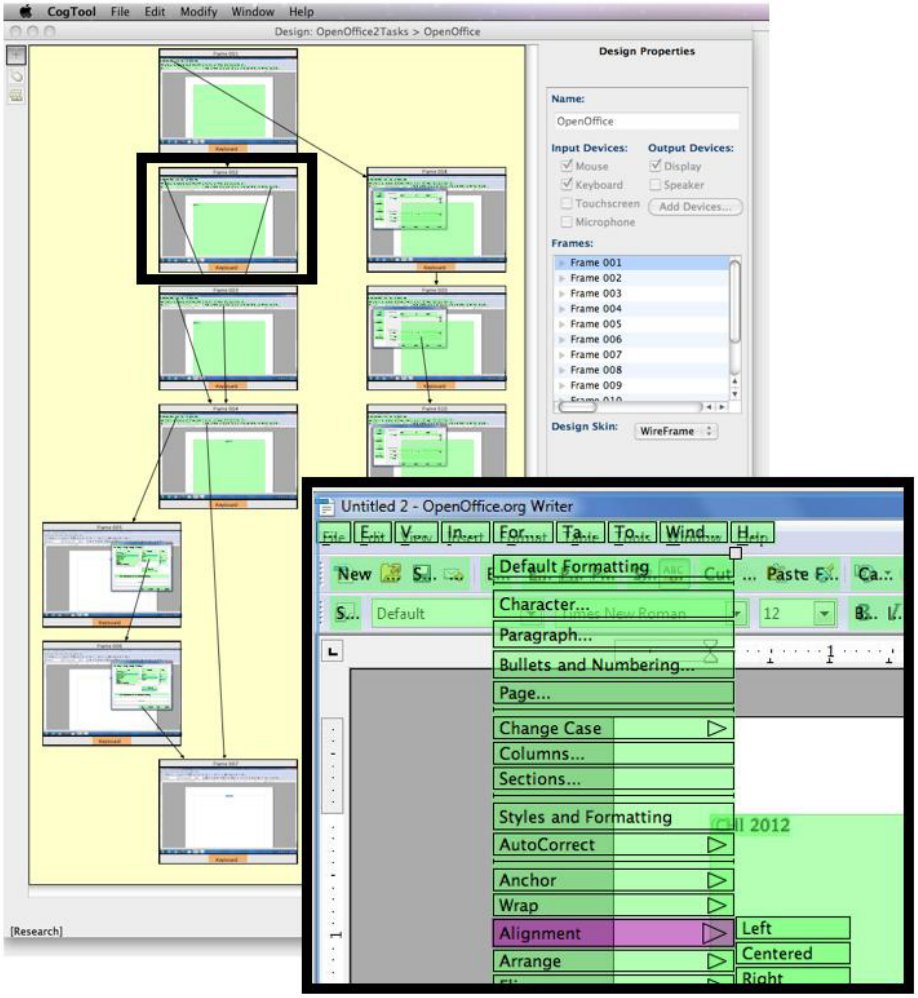


Figure 3.5: The imported CogTool design and one frame

Figure 3.4 shows the imported design, tasks, methods, and predictions and Figure 3.5 shows the completed CogTool design storyboard. The frames used in the "Enter, Center, and Bold Text" task run down the left side. The frames for the

"Insert Hyperlink" task run down the right side and share no frames with the other task except the first. A portion of the second frame in the storyboard shows part of the menu structure and toolbar widgets constructed by CogTool-Helper. The Format menu is expanded (occluding other widgets) and the Alignment item is selected, revealing the Centered item used in the Use Menus method. At this point, a UI designer can edit or add to the CogTool project as they can with a manually created CogTool project.

3.4 Method Inference

There may be alternative methods possible in the design that were not explicitly specified by the UI designer. CogTool-Helper uncovers these alternative methods and creates scripts for them so the UI designer can determine if their existence is a problem or an opportunity for the end user. This would be intractable if the UI designer had to manually create scripts for every possible path in the design. The method inference process generates all possible alternative methods for a task based on the frames and transitions created from the methods specified from the UI designer. Figure 3.6 is a schematic of the frames in the "Enter, Center and Bold Text" task in our example of using CogTool-Helper (3.1). In Frame 3, some text has been typed into a document in OpenOffice Writer and is selected. The two paths shown on the left (red dotted line and black solid line) represent the two methods for centering and bolding the text that were created by the designer using CogTool-Helper, but there is nothing preventing an end user from taking different paths through the design. The right side of this figure shows two such paths (thin gray line and thick blue line), where the model will switch from using the toolbar buttons to using the menus (or visa versa) to accomplish the task. We call these *inferred methods*. The last part of the Design

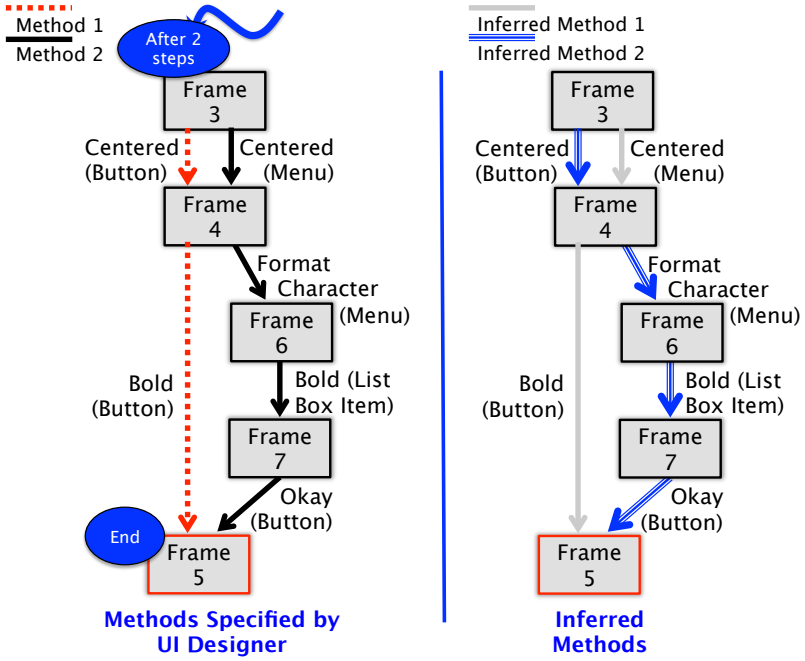


Figure 3.6: Inferred methods schematic

Construction phase is to calculate and create all of the possible inferred methods for a task.

To compute these methods, we extract the portion of the design corresponding to a single task, and then build a directed graph where the nodes are the frames and the edges are the transitions. The graph is a multigraph because two nodes can be connected by more than one edge.

3.4.1 Inferred Methods Algorithm

We use a depth first search algorithm to traverse the directed graph just discussed from the start to end node (frame), storing each edge (transition) that we visit along the way. Once we reach the final node (the end frame), we check to see if the path we have currently followed is in the set of paths we already have collected. If it is not, then we save this path as an inferred method, and create a CogTool method script

for it. We also add it to the set of collected paths. Once we have created all of the inferred methods, CogTool-Helper has finished and we can import and analyze the CogTool Designs that we have created.

Algorithm 1 Inferred Methods Algorithm

```
source: The source node for the path
destination: The destination node for the path
P: A set of edges representing a path from source to destination
paths: The set of paths we have already collected

function MethodInference(source, destination, P, paths)
    if source == destination then
        if paths does not contain P then
            add P to paths
        end if
    end if

    for all edges e in source.incidentEdges do
        vertex ← e.destination
        newPath ← P.copy
        if vertex is unexplored then
            add e to newPath
            recursively call MethodInference(vertex,destination, newPath)
        end if
    end for
    return paths
end function
```

Algorithm 1 describes the algorithm used to compute the inferred methods. The algorithm defines four variables:

1. *source* : This represents the source frame we are starting from in the task.
2. *destination* : This represents the destination frame we want to reach with the inferred method. This is the last frame reached by each of the methods in the task.
3. *paths* : This is the set of methods we have already collected, either through method inference or through the test cases replayed in CogTool-Helper.

4. P : An empty path which will hold the transitions leading from *source* to *destination*.

The algorithm calls the function *MethodInference* with these four arguments. If *source* equals *destination*, this means we have reached the destination frame. If *paths* does not contain the current path P, then we add P to *paths* as an additional inferred method. Then we go through each of the edges (transitions) incident to the current node (frame). If the node reached by the edge is unexplored in the current path, then we add this edge to the current path and call *MethodInference* again.

This algorithm is a modified depth-first search algorithm, and traverses all of the possible simple paths that do no repeat any nodes between the *source* and *destination* nodes in the graph. The complexity of this algorithm is in $O(|v| * |e|)$ where v is the number of vertices (frames) and e is the number of edges (transitions). In the worst case, the graph could be complete. The number of edges e would be equal to $v(v-1)/2$, so the worst case complexity of the algorithm is in $O(v^3)$. In all practicality however, we can expect the performance to be much better because the graph is very unlikely to be complete in our scenario. Also, it is unlikely that the number of nodes we would every have to traverse in our scenario would ever be more than one hundred because the tasks the designers will analyze with our tool are unlikely to be that complex, so we do not expect the complexity of this algorithm to become an issue.

Figure 3.7 shows the CogTool project window for the task "Enter, Bold, and Center Text". The three methods that were input to this project are "Using Menus", "Using Keyboard Shortcuts", and "Using Toolbar Buttons". In addition, CogTool-Helper found nine inferred methods. These are labeled *Method (Inferred) 01*, *Method (Inferred) 02*, etc. These were found in the design created by replaying the first three test cases (methods).

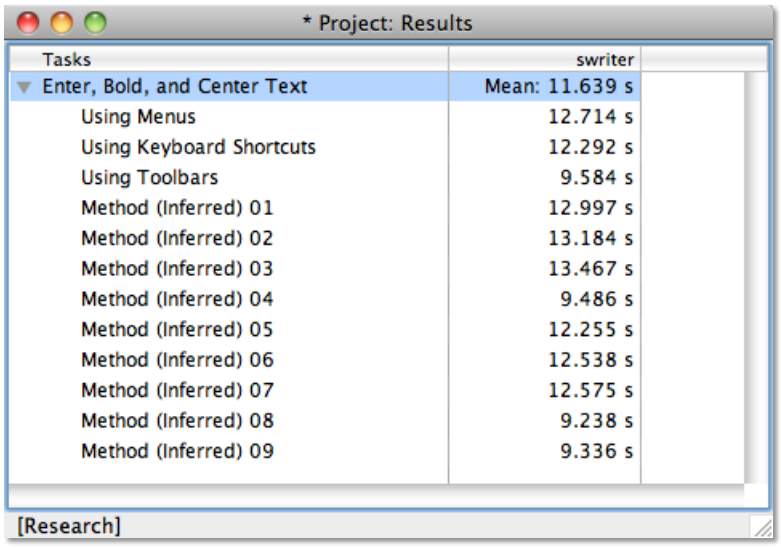


Figure 3.7: Project window for Enter, Bold, Center task with 9 inferred methods

3.5 Supported Application Types

CogTool-Helper currently works in two GUI environments. Since different application platforms have slightly different Accessibility APIs, we have to use different versions of the GUITAR framework for each environment and then customize the CogTool script creation. To date, our tool works on applications written in Java using the JFC platform and on those that use the Open Office (UNO) API [40]. Office productivity suites that support the UNO API are OpenOffice [39], LibreOffice [23], and NeoOffice [38]. We have tested CogTool-Helper with two of these suites; OpenOffice and LibreOffice. Within OpenOffice, we have tested CogTool-Helper on each of the OpenOffice 3.0 applications, Impress, Calc, Math, Base, and Draw in addition to Writer, which was used for the extended example in this thesis. Additionally, we have successfully used CogTool-Helper with LibreOffice 3.4.3. Modules tested in LibreOffice include Writer, Impress, and Calc.

For the Java JFC implementation of CogTool-Helper, we have tested TerpWord and TerpSpreadsheet from the TerpOffice Suite, a set of Office Productivity tools written at the University of Maryland [29].

CogTool-Helper requires Java 1.6 and has been tested on a Windows 7 operating system. In addition, we have successfully used CogTool-Helper in a Mac OSX (Version 10.6.8) environment and a Linux environment (2.6.18). We use CogTool-Helper in the Linux environment in our feasibility study described in Chapter 4. In MacOSX and Linux, we have tested CogTool-Helper only with the LibreOffice application.

3.6 Technical Limitations

The goal of this work is to make the entire process of creating the models completely automatic, however, there are parts of our current implementation that are only semi-automated. The first part is task definition. The designer can either demonstrate the task by hand on the interface (which is automatically captured by CogTool-Helper), or provide test cases in the GUITAR format. Demonstration would be entirely manual, but Chapter 4 presents a method for generating test cases for a task that can be then imported into CogTool-Helper.

The design construction process is entirely automatic. The designer does not need to interact with the application at all during this stage. However, once the design is created, the designer must import this by hand into CogTool (which is done by a simple menu action).

Automating the connection between CogTool-Helper and CogTool with the new design imported is left as future work. CogTool does not yet have a command line method for importing projects from an XML file.

Although we have been able to create CogTool designs for all of the systems described in Section 3.5 above, we have also identified a set of limitations. First we are currently constrained by the capabilities of the underlying GUITAR testing tools. The action handlers provided by GUITAR are limited so we can support only a small number of actions. These do not include right-clicks, double-clicks, drag actions, and more complex actions we would like to support. GUITAR was not explicitly built with the intention to mimic a human so many of these actions have been considered unnecessary. We intend to work with the GUITAR team in the future to expand the set of actions we have available to perform. GUITAR did not include the ability to perform keyboard shortcuts, which we have added as an addition to GUITAR.

Another limitation is that we have mapped only a subset of accessibility types to their appropriate CogTool widgets. These cover the majority of widgets in the Java and OpenOffice systems, but we have not attempted to be complete, for instance, we do not yet handle combo buttons (where the button drops down a menu) or context-menus (where a right click on a widget opens up a menu). Future work will be to expand our set of widget translations.

Finally, we have implemented the task capture feature for the OpenOffice applications only. We plan to implement this for Java JFC applications as well. We would also like to expand the set of supported application types for CogTool-Helper to all of those that GUITAR supports.

As additional future work, we will add the ability for CogTool-Helper tasks to start in different states of the application and we will optimize the time it takes to build a design. We will also support designs where the menu structure can change dynamically as the application runs, which may require an alternate technique beyond capturing the menu only once at the start of our process.

3.7 Summary

In this chapter, we presented CogTool-Helper, a tool to bridge the gap between the GUI testing community and the human performance modeling community in HCI. We presented details of CogTool-Helper's strategy and its implementation and our algorithm for computing inferred methods. We believe that CogTool-Helper is most useful for comparing legacy applications as a baseline for new design ideas and will allow the user interface analyst to spend more time analyzing their designs than constructing models of them.

Chapter 4

Towards Human Performance Regression Testing

Regression testing has become best practice in the development of commercial software. Tools and automated processes have been developed to solve problems related to regression testing for functional correctness; testing after a change has been made to detect a fault introduced by the change [42]. Particular quality attributes, such as system response time [49, 51], have been studied, but testing for usability has largely been ignored. As systems grow larger, they often become more complex, hurting end-user efficiency.

When testing new features out on an existing system, there are no skilled users for the features. Tools for predictive human performance modeling, such as CogTool [18], can simulate end-users [12, 18] and test out these new features before skilled users exist [7]. Since regression testing is typically resource constrained [6, 42], the manual effort required to perform usability testing means that regression testing for usability is often intractable. CogTool-Helper [47] can eliminate much of this manually-intensive process. Although CogTool-Helper is useful for analyzing existing systems

as is, it is not useful for automated regression testing of human performance because of several limitations. First, designers must manually encode test cases representing their tasks to import into CogTool-Helper, or capture the methods by hand. This is a huge limitation on the number of methods for a task that can be analyzed. Also, if the designer does not capture or encode methods performing the task in multiple orders, alternate orders will not be seen in the final design. Last, test cases written on one version of the interface may not run on the next; for example, if the test case contains events on widgets that have been removed. The test cases will need to be adjusted to remove changed widgets or add new widgets.

To move toward automated human performance testing as a normal regression testing activity, we have extended CogTool-Helper to generate test cases that can be used for human performance testing. This will allow a much larger and diverse variety of methods for tasks to be analyzed and more tests to be run across a wide range of scenarios. We do this using currently existing GUI testing tools by defining a simple set of rules to govern the task. We analyze the usefulness of CogTool-Helper and our CogTool-Helper extension for generating meaningful test cases ,and find that it can provide meaningful information to both UI designers and regression testers. We also demonstrate that we can improve the efficiency of CogTool-Helper through sampling of test cases that relies on CogTool-Helper's ability to uncover implicit methods for a task beyond what the CogTool-Helper user specifies.

4.1 Motivation

In Chapter 3, we did not discuss how to decide which test cases to import or demonstrate in CogTool-Helper, but assumed the UI designer would bring their knowledge of frequent or important tasks to bear on that selection. In functional GUI testing,

test cases are selected or generated with respect to their functional capability, but in human performance evaluation test cases must be semantically meaningful from an end-user's point of view. For example, it makes sense in functional GUI testing to use test cases that include clicking on the same button twice because hidden problems such as incorrect initialization, undetected on the first click, would be revealed through such a test case. However, evaluating such test cases for efficiency of task execution by skilled users would not be worthwhile since accomplishing a real-world task would not include superfluous repeat button presses. Thus, we will use functional test case generation techniques and tools, but constrain them to produce only test cases meaningful to the end-user.

In the work of Memon et al. [32], AI planning was used to generate GUI test cases for particular tasks. Operators are defined to perform atomic actions such as `open a file`. Each operator is a set of one or more events (e.g. *file*, *open* are two events that make up `open a file`). Plans are created that consist of the desired start state, the desired end state, the task written as a partial order of operators, and a set of ordering constraints that refine these orders. (Events that perform only structural actions, such as opening a new window or pulling down a list, are not included in the plan). The resulting plans are then passed to a test generation algorithm that uses a planner to generate an instance of the test case. While this work is very similar to ours, it requires the tester to define start and end states for each part of the plan and to specify in a procedural manner the exact operators within the plan. It also restricts the planner to system interaction events which may miss structural alternatives on the interface that can impact user performance (e.g., whether an event is invoked by clicking the mouse or with a keyboard shortcut). In addition, although more than one test case can be generated for a single plan, the aim is not an exhaustive set of test cases that can complete the same task on an interface.

The goal of our work is similar, but differs in several key ways. First, since users can often accomplish tasks in many different ways, we want to specify the task with as few orderings as possible, moving away from the strictly procedural view of a task. Second, we want the UI designer to be able to explore differences in user behavior caused by differences in structural events. For instance, if a user utilizes the toolbar to make text bold, this presents different human performance results than making text bold by opening the Format menu.

In this work, we utilize the GUITAR testing framework [30] as our exemplar for this process, because we already use its replay mechanism as the technical implementation for CogTool-Helper. In this section, we present details of its implementation when describing our process, but point out that we believe the process is general and could be developed using other tools as well.

4.2 Process Overview

Figure 4.1 shows an overview of our extension to CogTool-Helper. The bottom portion of this figure is our existing process for CogTool-Helper. Since CogTool-Helper includes an import feature (Step 2), we can pass generated test cases from our new process into this phase. As in a regression testing scenario, the process in Figure 4.1 is performed on each successive version of the system that includes an interface modification. The first step in generation (Step A, top of Figure 4.1), identifies widgets and their actions that form the full set of events that may be used to achieve our task. For instance, we may include both menu items and keyboard shortcuts; alternatives to achieve the same part of a task. Step B creates an EFG that contains only those events and their relationships identified in the first step. Step C defines a set of rules

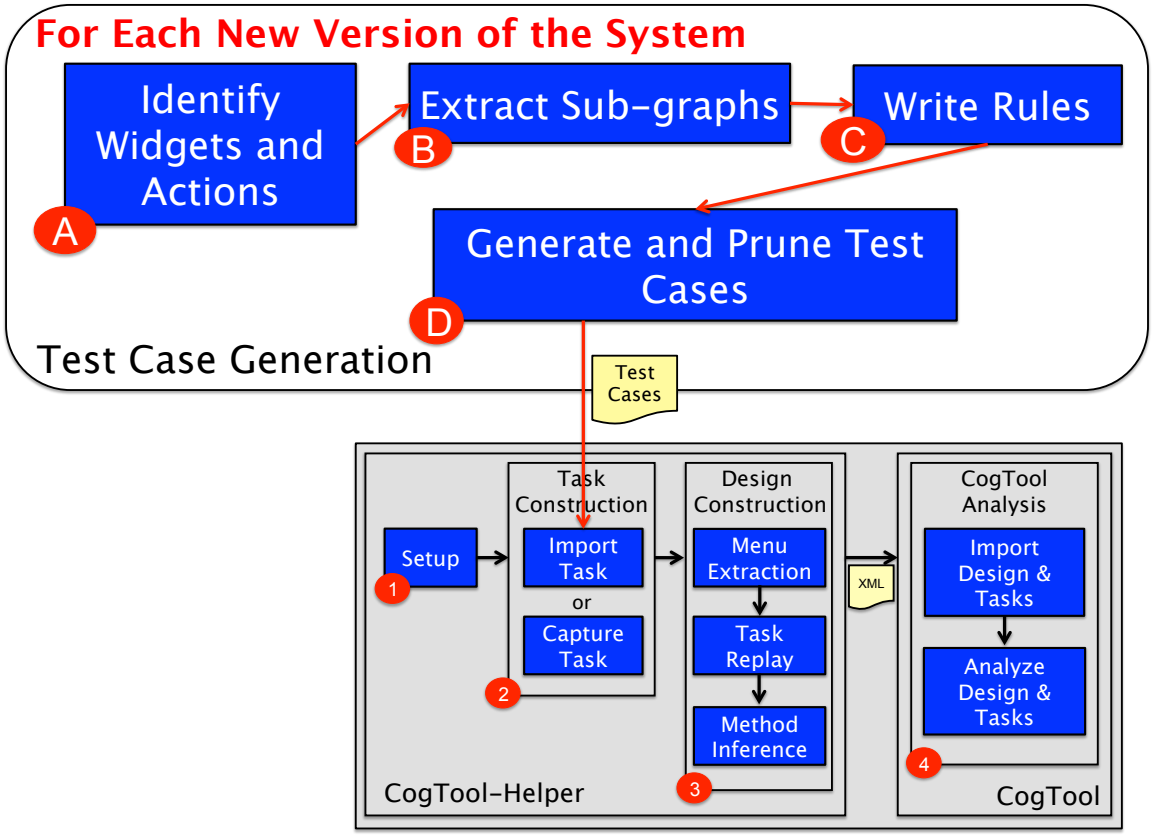
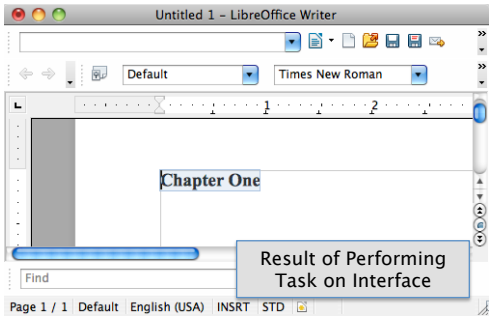


Figure 4.1: Human performance test generation workflow

that refine the task to make sense semantically. Step D uses the EFG combined with the rules to generate all possible test cases that perform this task.

4.2.1 Identify Widgets and Actions

To explain our test generation process, we use the task given in our scenario of Sally, the UI designer. Her task is to type in the text `Chapter One`, select the text, and make it bold and centered. We leave out the "centered" action here for simplicity. We use the open source office application, LibreOffice [23], to demonstrate this process. The resulting state of this task is shown in the top left of Figure 4.2.



Sub-Goal	Approach	Partial Event Tuple: <Name, Type, Action>
Type Text: Chapter One		1. <..., PARAGRAPH, Typing>
Select All of the Text	A . Toolbar B. Menu C. Keyboard	2. <Select All, PUSH_BUTTON, Click> 3. <Edit, MENU, Click>, 4. <Select All, MENU_ITEM, Click> 5. <Select All, MENU_ITEM, Keyboard Shortcut>
Make Text Bold	A. Toolbar B. Menu	6. <Bold, TOGGLE_BUTTON, Click> 7. <Format, MENU, Click >, 8. <Character..., MENU_ITEM, Click> 9. <Bold, LIST_ITEM Select_From_List>, 10. <Okay, PUSH_BUTTON, Click>

Figure 4.2: Example task in LibreOffice Writer

In general, UI designers will evaluate many tasks on an interface based on their knowledge of their user-base, including tasks far more complex than this one. This task is limited for illustration purposes, restricting our analysis so that the user types in the text *first* rather than starting the task by setting the font to bold. Most tasks will not be as restrictive. For this example, we assume a version of the software that has menus, keyboard shortcuts, and toolbars; any of which, in different combinations, can be used to perform this task.

We use an event-tuple to represent the information required for test case generation with properties that GUITAR uses to identify and perform actions on the interface [31]. The event-tuple takes the following form:

<`Title (T), Class (C), Window (W), Action (A), Parent (P), Parameter (R)`>

`Title` is the textual label of the widget. If the widget does not have a textual label, such as a button that only displays an icon, but it has a tooltip, then the tooltip text is used. The Title property helps GUITAR identify the widget on the interface. `Class` describes the type of widget, such as a PUSH_BUTTON, TOGGLE_BUTTON, etc. These types correspond to those extracted by GUITAR for the Class property

and differ based on GUI framework. The Class property also helps GUITAR identify the widget.

`Window` is the textual label of the window containing this event-tuple; also needed for widget identification. `Action` defines which GUITAR event handler will be used for this event. Its values include *Click* (currently we only support left click), *Typing* (to represent typing with the keyboard), *Set Value*, *Select from List*, *Keyboard Shortcut* and *Keyboard Access*. *Keyboard Access* is used when the keystrokes walk through a hierarchical menu instead of directly accessing a command (e.g., `Alt-oh` opens the FOrmat menu and selects the CHaracter item). The implementation of GUITAR available to us did not provide keyboard shortcuts, so we have added these to our version. These actions and their corresponding GUITAR action handlers are detailed in Appendix A.2.

`Parent` is optional. It is the `Title` of the container for this event-tuple. Parent provides a way to disambiguate information when more than one widget in a window matches the same `Title` text, or when a widget does not have any `Title` text. `Parameter`, also optional, is only used for widgets with the action `Typing` to tell GUITAR what text to type. CogTool-Helper supports several actions for typing text, *insert*, *replace*, *select*, *unselect* and *cursor*, some of which have additional parameters beyond these six, such as the text to insert or replace. These are also detailed in Appendix A.2.

As some of these properties are not easy to gather directly from the interface without knowledge of how GUITAR works, we envision a user experience professional describing the task , and the testing professional, who is more familiar with the testing tools, would identify these tuples for each event. The event-tuples for our UI designer Sally's task are shown on the right side of Figure 4.2. The event-tuple has been reduced to show only <T,C,A> since they are enough to make each event

unique in our example. The first column in the table is a task subgoal. The second column lists the approaches that would lead to the events (e.g., using the menu or keyboard). The last column shows the event-tuples associated with each approach. In our example the main paragraph widget for the document has no name (Event Tuple 1). This is a situation where we would use the optional `Parent` parameter, which would be, in this case, the text for the main document window , "Untitled 1 - LibreOffice Writer". Referring back to Sally's case, we envision she would define the sub-goals and approaches she wants (Columns 1 and 2 in Figure 4.2), and the testing professional, who we call Joan, would proceed to identify the event-tuples shown in Column 3.

4.2.2 Extracting the Sub-Graphs

After gathering all of the event-tuples for the task, the list is passed to a *filter* that we wrote to plug into the GUITAR ripper. The filter reads in the list of event-tuples in an XML format. As the ripper finds a widget, it searches the passed in list for the widget. If the widget is found in the list of event-tuples, it rips the corresponding action in the tuple and performs that action on the interface. If the widget is not found in the list, it rips only a basic set of information for that widget and does not perform any actions. This way, only the areas of the interface that contain widgets used in the task are ripped.

After running the ripper, the result is an EFG containing only those events included in the input set. Since most EFGs for real applications are very large, (The OpenOffice Writer 3.3.0 interface has 605 nodes and 79,107 edges [30]), this filter substantially reduces the number of events and provides a tractable space within which to work. The EFG representing our example task is shown in Figure 4.3. In this EFG

we see the ten events (nodes) corresponding to the table in Figure 4.2 and 52 node relationships (edges). We have also shown information on this EFG about the *type* of event for each node, encoded by GUITAR as a structural property of the node [31]. These are presented in Chapter 2, but we repeat them for convenience here. A **System Interaction** event causes a functional change to the system (e.g., selecting all the text). The rest of the event types cause structural changes. An **Expand** event opens a menu or a list to allow the user to select from a set of options. A **Restricted Focus** event opens a modal window; the user must interact with that window until it is closed. An **Unrestricted Focus** event opens a non-modal window. We do not have any unrestricted focus events in our example task. Finally, a **Terminal** event closes a window.

4.2.3 Defining Rules

A test case is defined as a path through an EFG, often restricted to a specific length. However, even with the EFG reduced to nodes relevant to a particular task, not every path makes sense for human performance regression testing. For example, referring to paths of length three possible in Figure 4.3, the path with events [#1, #5, #6] (selecting text and making it bold using the toolbar buttons) is semantically meaningful as the representation of a skilled user's behavior when making the text bold. However, [#3, #3, #3] (three clicks on the Edit Menu) is valid for functional testing, but does not make the text bold.

We would like to restrict the generator to generate only test cases that perform the task. To do this, we constrain the generator with two kinds of rules. The first is a *global rule*, enforced for all tasks. Global rules stem from typical skilled user behavior, apply to most tasks, can be written in advance of any particular project or

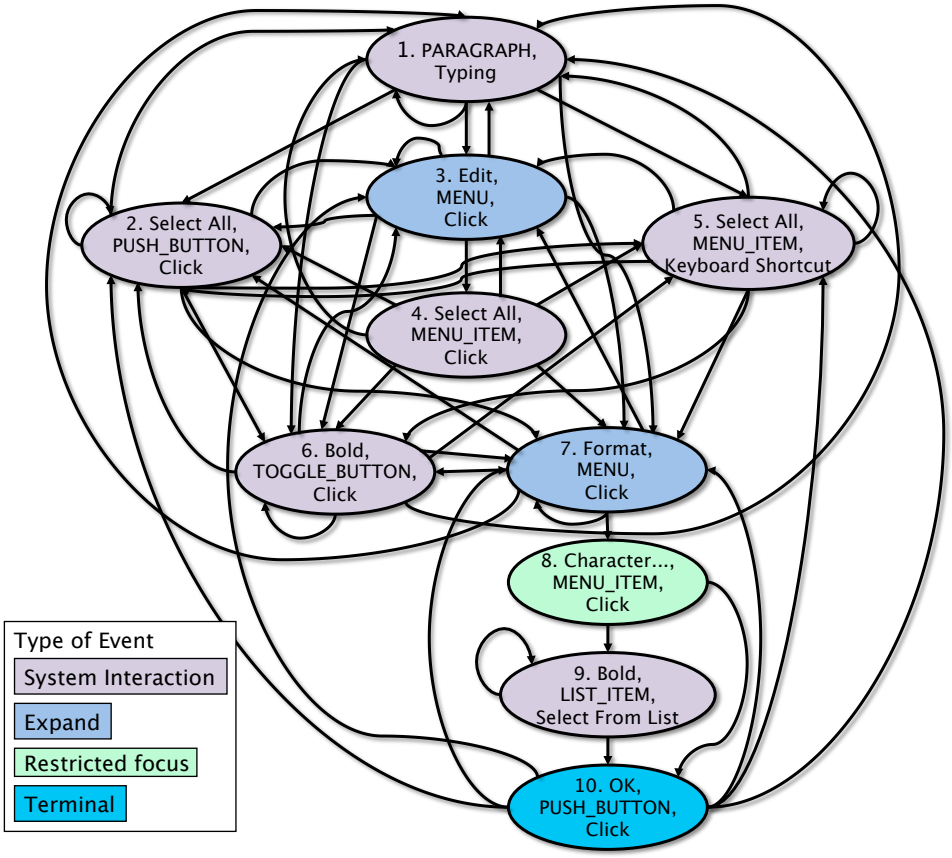


Figure 4.3: Resulting Event-Flow Graph for Sally's task

task, and can be reused. The second kind of rule is *task specific*. These arise from the logic of the specific task and how it should be performed and therefore need to be written anew for each task and/or interface to be analyzed. Global rules rely on the GUITAR event type (e.g. System Interaction, Terminal) and the GUI structure. Task specific rules can override global rules if the task and UI so require.

4.2.3.1 Global Rules

We have currently defined four global rules that we apply to all tasks used in our feasibility study.

1. *End in Main Window*. The test case must end with a system interaction event in the main window, or with a terminal event resulting in only the main window being opened. Therefore, expand events cannot be last in a test case because they open a menu or list. A restricted focus event cannot be last because it opens a new window. Also, any event inside of a window other than the main window cannot be the last event in the test case. Events #3, #7, #8, and #9 cannot be last events in a test case.

2. *Expand Followed by Child Event*. An event that expands a menu or list must be immediately followed by an event that is executed on a child of that menu or list. It prevents the test case from expanding a menu or list and then performing no actions on it. For example, after event #3 (Edit, MENU, Click) is performed, the only valid event that can follow in this graph would be #4 (Select All, MENU_ITEM, Click). There is an edge on this graph leading from #3 to #7, but this edge cannot appear in any test case.

3. *Window Open and Close Can't Happen*. A window cannot be opened and immediately closed without some other event happening in between. In our EFG we cannot have event #8 immediately followed by #10, despite an existing edge, because it would result in a meaningless user action. This rule will force the test case generator to take the path from event #8 to #9.

4. *No Repeat Events*. The last rule is used in combination with a task specific rule. It states that no event can appear more than once in a test case unless it appears in a local rule that overrides it. The user cannot click event #6 (make the text bold) two times in the same test case. We will explain why this rule may be overridden shortly.

4.2.3.2 Task Specific Rules

The global rules are primarily structural, enforcing constraints that are necessary for all tasks. However, individual tasks and UIs also have constraints that restrict them based on their functional properties. Task specific rules ensure that the set of events achieves the required task. We have defined four task specific rules.

1. *Exclusion*. This is a mutual exclusion rule. It says that exactly one of the events included in an exclusion set must be included in each test case. Events included in an exclusion set achieve the same goal. Examples of events that would be in an exclusion set for our example task are #2, #4, and #5 because they achieve the same goal of selecting the text.
2. *Order*. This rule specifies a partial order on events. We group events into *Order Groups* , i.e., sets of events that are in the same ordering equivalence class, and then place the groups in the required order. Only system interaction events need to appear in the Order rule, since the other types of events only cause structural changes. In our UI designer Sally's task, we required typing the text to happen before all other events. We would place #1 (PARAGRAPH, Typing) alone in the first order group. Since selecting the text must happen in this UI before it can be made bold, we place events #2, #4 and #5 in the second order group. Finally, making the text bold, events #6 and #9 are in the last order group.
3. *Required*. Events in the required list must appear in all test cases. In our example the only event that is required is event #1 (PARAGRAPH, Typing).
4. *Repeat*. Events in the repeat list allow us to include specific events in a test case more than once, overriding the global rule *No Repeat Events*. We don't have any repeat rules in our example task; however, if our task also italicized the

text, then we would would need to allow #7 (Format MENU, click) to appear more than once since a task that performs both bold and italic using only menus may need to expand the same menu more than once.

4.2.4 Generating Test Cases

Once we obtain the EFG and the set of rules are written, we supply these as input to an existing test case generator and generate all possible test cases for this EFG that are valid with respect to the rules. As the generator creates test cases, it checks each one against the rules and prunes out those that do not pass. We currently use the existing GUITAR test case generator (Sequence Length plugin) that accepts an EFG and test case length as a starting point. Using our knowledge of the task, we determine all possible length test cases that will reasonably be used by a skilled user to perform the task. We input those lengths into the test case generator and generate all possible test cases for that length that abide by our rules. We have implemented additional functions into the test case generator to check the rules. The generated test cases are then fed into CogTool-Helper which turns each test case into a CogTool method of how a skilled user might perform this task.

4.3 Feasibility Study

We conducted a preliminary study to determine the feasibility of our approach. In the first part of this study (RQ1), we generate all possible test cases for the task so the number of test cases and CogTool methods are equal. In the second part of the study, we evaluate our ability to optimize the run-time of CogTool-Helper and select a subset of the generated test cases to run in CogTool-Helper. We use CogTool-Helper's

method inference process to add to the set of methods corresponding to the generated test cases.

4.3.1 Research Questions

We consider the following two research questions in our feasibility study.

RQ1: Can automated user model creation using CogTool-Helper provide useful information for human performance regression testing?

With this question, we examine the usefulness of the information CogTool-Helper provides in a regression testing scenario. We seek to examine whether the information provided can help UI designers, analysts, and product teams quantitatively validate the addition or removal of certain interface features in terms of user performance and use this information to improve their interface designs.

RQ2: What is the impact on cost and effectiveness of sampling test cases?

This question examines the impact on cost (time taken to run CogTool-Helper), and effectiveness (the value of the information CogTool-Helper can provide) by sampling test cases from the entire set generated for the task. This question evaluates the ability of CogTool-Helper to infer methods from an existing design and set of methods for a task.

4.3.2 Feasibility Study Scenario

Consider the situation of an organization that produces office productivity products, like word processing, presentation and spreadsheet software; for example, where our UI designer, Sally, and testing professional, Joan, work. In Sally and Joan's company, products will often go through several versions, often with changes to the UI as well

as functionality. The product teams will want to know if the changes are making their users more productive or not, information they can also use in their marketing and sales materials. This feasibility study tests whether automatic generation of test cases for the purpose of usability evaluation (i.e., efficiency) can provide that information.

We selected three modules of LibreOffice 3.4 [23], `swriter`, `simpress` and `scalc`, to illustrate the process the product team would go through and the resulting information it would gain from user performance regression testing.

The first step our UI designer Sally takes is to identify tasks that the end-user would do in the real world and create representative instances of those tasks. This information usually results from field studies, interviews, questionnaires, or log reports interpreted by user experience professionals. As an illustration, we selected four tasks for our study: two tasks in `Writer` (*Format Text* and *Insert Hyperlink*), one task in `Impress` (*Insert Table*) and one in `Calc` (*Absolute Value*). We provide brief descriptions of the tasks in Table 4.1. Detailed descriptions of these four tasks, their event-tuples, and the set of rules we applied to generate test cases can be found in Appendix B.

Our study considers three hypothetical versions of LibreOffice that introduce different UI features to the end-users. The first version (M) presented only menus to access the functions needed for these tasks. The second (MK) added the ability to access these functions with keyboard shortcuts. The third (MKT) added toolbars for common functions (the default appearance of LibreOffice 3.4).

4.3.3 Metrics

The quantitative predictions of skilled task performance time for each method on each version of the system, and the resulting distributions of those predictions, will

Table 4.1: Tasks Used in the Study

LibreOffice Module	Task Name	Task Description	Version	No. Events	No. Rules
Writer	Format Text	Text *Google* typed in, selected, and made bold and centered	M	9	4
			MK	12	5
			MKT	13	7
	Insert Hyperlink	Insert Hyperlink to *Amazon*, and make text of link uppercase.	M	9	3
			MK	11	5
			MKT	13	8
Calc	Insert Function	Insert absolute value function, shift cells to right, and turn of column & row headers	M	11	4
			MK	14	6
			MKT	16	10
Impress	Insert Table	Insert a table, add a new slide, and hide task pane	M	7	3
			MK	9	5
			MKT	11	7

speak to whether automatic test case generation would produce interesting results for UI usability evaluation (*RQ1*).

For *RQ2*, the metrics are the run time required to run the test cases in CogTool-Helper, the total number of methods resulting from these test cases in the final CogTool project, the number of inferred methods added by CogTool-Helper, and the human performance predictions for all of the methods.

4.3.4 Study Methods: RQ1

To simulate the first two hypothetical versions of LibreOffice in our study, we simply removed the toolbars using LibreOffice 3.4's customization facility. Once the tasks and versions are selected, a testing professional (Joan) and a user experience professional (Sally) together would follow the procedure detailed in Section 4.2.

First, Sally, the UI professional, would identify the widgets and actions she wants in the task. Then Joan, the testing professional, would encode them in GUITAR's format. In our case, one person fulfilled both of these roles, identifying the list of event-tuples and encoding them, resulting in the number of events listed for each task

and version of the system shown in Table 4.1 (the complete list of events and rules can be found in Appendix B).

Joan and Sally would then review the global rules together and decide if they applied to their system or if a local task rule would need to be written to override a global rule. Together they would write local task rules to express the constraints of each task, again combining their complementary knowledge of the real-world tasks and GUI testing syntax. Our set of rules defined for each task, for each tool version, are shown in the rightmost column of Table 4.1 (and detailed in Appendix B).

Next, Joan would run the test case generator against the rules to generate the test cases. She would import these into CogTool-Helper and run it to obtain the CogTool project files. Then Sally, the user experience professional, would import these project files into CogTool to obtain the human performance predictions detailed in the next section. For this study, we created a command-line version of CogTool-Helper to automate the import and task construction phase, and then we imported each project file into CogTool.

4.3.5 Study Methods: RQ2

To investigate the impact on cost and effectiveness of sampling test cases (RQ2), for each task, we use the last version, with menus, keyboard shortcuts and toolbars (MKT) since those have the largest number of test cases. We randomly select (without replacement), the required number of test cases for 5, 10, 25, 50% of the complete set of test cases. We sample five times at each percentage for each task, to prevent bias from a single outlier run. We then run CogTool-Helper on the samples of test cases and capture the number of methods in the final CogTool project, the number of inferred methods, the run time required to create the designs, as well as the human

performance predictions for all of the methods. We then report averages of these values in our results.

4.3.6 Study Environment

We ran all of our experiments on a 64 bit Linux cluster where each cluster node contains 32 Opteron 6128, 2.2. GHz processor cores, and 128 GB of RAM. The grid is running Linux 2.6.18 and Java 1.6.0_10. We used the Xvfb program to enable us to run our applications without a screen. Each set if test cases was run in CogTool-Helper using LibreOffice version 3.4.

For test case generation, we used the GUITAR framework (UNO version), version 1.3, updated from the svn repository in January 2012. We modified the ripper, replayer and test case generator for our experiments. First we added event handlers to the replayer for performing *keyboard shortcuts* and *keyboard access*. Second, we added a filter to work with the ripper, to tell it to extract the reduced EFG as described in Section 4.2. We did not modify the module that creates the EFG at all. We also added a module to check rules during test case generation (also described in Section 4.2).

4.3.7 Threats to Validity

In every study such as ours, there are certain threats to the validity of the study. Shull [45] et. al. defines four types of threats threats to validity that can have an impact on the validity of a study: *Conclusion*, *External*, *Internal*, and *Construct*. We provide definitions of the last three types here and describe how we have addressed these threats.

External Validity refers to the degree with which the findings of the study can be generalized to other participant populations or settings.

The primary threat to external validity is that we have run our experiments on only a single application, LibreOffice, and using only four simple tasks. It is unclear whether our results will generalize beyond the simple tasks studied to more complex tasks, or to other applications. However, LibreOffice is representative of a family of office software and does not have any unusual GUI features that would make it particularly amenable to our study. We also evaluated tasks on three different modules within LibreOffice and selected four tasks that are quite different.

Internal Validity refers to whether an experimental treatment/condition makes a difference or not and whether there is evidence to support that claim.

It is possible that there is a bug in one of the steps of our test generation process which is a threat to internal validity. However, we have manually inspected the final CogTool designs for many of our results and they appear to be correct; we have randomly selected other artifacts for manual inspection. In each of the inspected designs, all methods for a task ended up in the same frame so it appears that all of our generated test cases actually perform the same task. It is also possible that some of the calculations by CogTool are incorrect, however that tool has been in use by hundreds of user experience professionals since 2004, so we believe that this is a minimal risk.

Construct Validity refers to whether specific measures model dependent and independent variables from which the hypothesis is constructed. An empirical study with high construct validity ensures the study parameters are relevant to the research questions.

Table 4.2: Human Performance Predictions: Skilled Task Execution Time (seconds)

Task (Version of System)	**No. Test Cases**	**Mean Time**	**Min Time**	**Max Time**	**SD**
Format Text (M)	3	13.4	13.4	13.4	0.0
Format Text (MK)	24	12.8	11.9	13.7	0.6
Format Text (MKT)	81	11.5	8.3	13.7	1.7
Insert Hyperlink (M)	2	20.5	19.4	21.6	1.6
Insert Hyperlink (MK)	8	20.1	18.3	21.6	1.4
Insert Hyperlink (MKT)	18	19.8	17.5	21.6	1.3
Absolute Value (M)	4	18.1	17.9	18.3	0.1
Absolute Value (MK)	32	18.3	17.7	18.8	0.2
Absolute Value (MKT)	72	17.8	14.1	18.9	1.6
Insert Table (M)	3	12.8	12.7	12.9	0.1
Insert Table (MK)	12	12.7	12.3	13.3	0.3
Insert Table (MKT)	36	12.3	11.3	13.3	0.4

As a threat to construct validity, it is possible that we could have measured other variables in our study. However, we believe the time predictions obtained from CogTool for performing a task provide the most valuable information to UI analysts. For our second research question, we measured the time taken to run the test cases in CogTool-Helper. We believe this is the most important factor that could be limiting to a UI analyst when using our tool, so it is the most relevant factor to measure for our second research question.

4.4 Results and Discussion

In this section, we provide data to answer our two research questions and discuss the implications of these results. We then discuss the practical application of our approach and future directions.

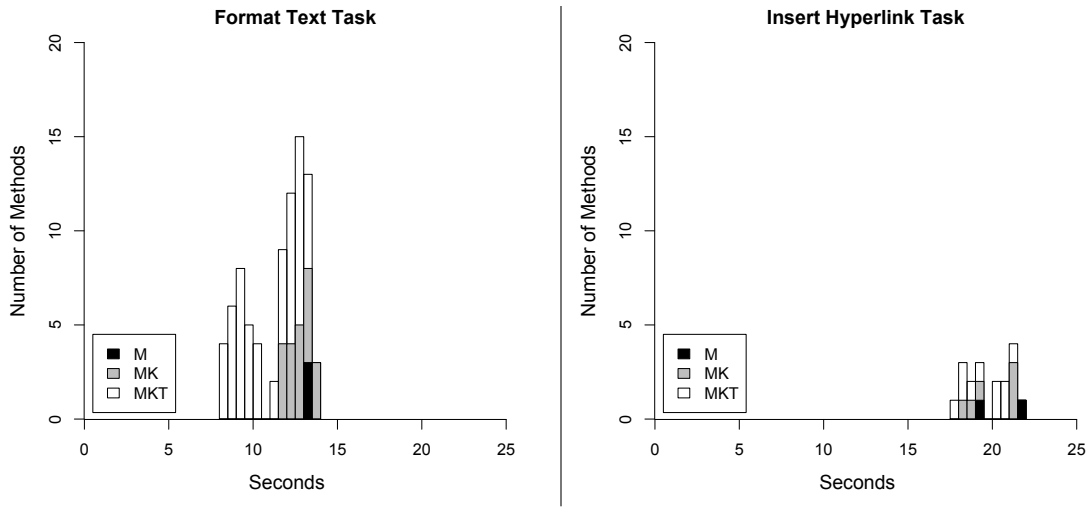


Figure 4.4: Histograms of Predictions of Skilled Task Execution Times (Writer)

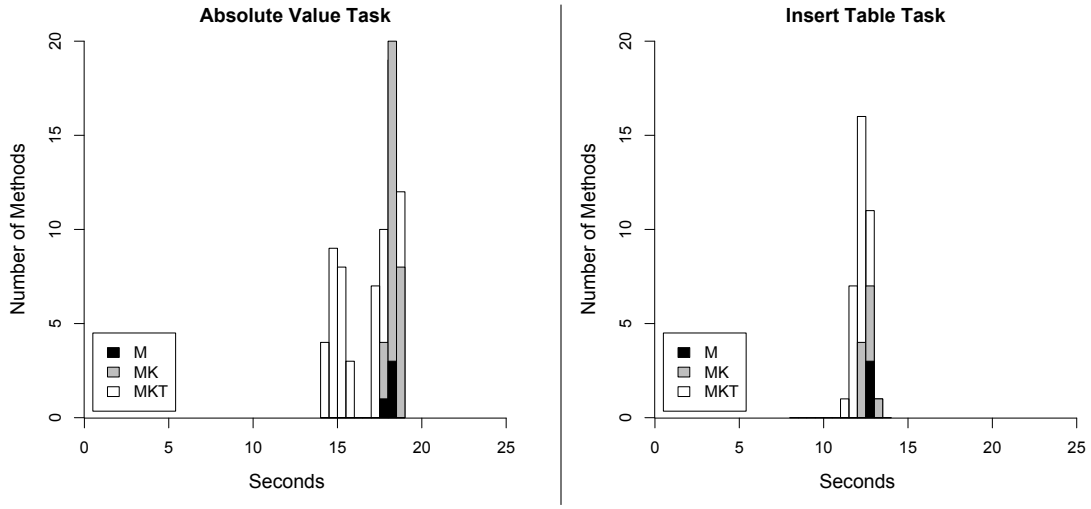


Figure 4.5: Histograms of Predictions of Skilled Task Execution Times (Calc and Impress)

4.4.1 RQ1: Usefulness in UI Regression Testing

Table 4.2 shows the three versions of each task: menu only (M), menu + keyboard (MK) and menu + keyboard + toolbar (MKT). For each version, we show the number of test cases generated, the mean time predicted for a skilled user to accomplish this

task, the minimum predicted time, the maximum predicted time, and the standard deviation. From the raw predictions, we show histograms of the number of test cases by time predictions for each task in each version of the system (Figures 4.4 and 4.5). These histograms are a major contribution of our work because, until this point, user experience professionals would model only one or two ways a user could accomplish a task because the modeling had to be done by hand.

We look first at Table 4.1. In all but one case (Absolute Value), the mean time decreases with the addition of keyboard shortcuts and in all cases it decreases again with the addition of the toolbar. This suggests that, for tasks that can take advantage of toolbars, the skilled end-user will indeed be able to perform their work in less time with the newest version.

Perhaps more revealing is the decrease in minimum time needed to accomplish each task, which in all cases decreases with system version, by as much as 40% for the *Format* task, and 21% for the Absolute Value task. This suggests that the most proficient skilled users could be substantially more efficient, information potentially useful for marketing or sales. In addition, the paths that displayed this efficiency might feed directly into training videos or "tips of the day" to help end-users attain such proficiency.

The maximum time tells a different story about the benefits of keyboard shortcuts and toolbars. In three of the four tasks adding the keyboard shortcuts *increases* the maximum predicted time because methods that mix menus and keyboards require the user to move the hand between the mouse and keyboard. This information might encourage project teams to increase the coverage of keyboard shortcuts to keep the user's hands on the keyboard (this would also have the side effect of increasing accessibility for users who cannot operate a mouse). On the other hand, adding the toolbar does not effect the maximum time for three of the tasks, meaning that no

mixed methods involving the toolbar are slower than those involving the menus and keyboard. Therefore, adding the toolbar has no downside for efficiency. However, in the *Absolute Value* task, the maximum time increases slightly because the presence of the toolbar forced a dialog box to be moved, requiring the user to move the mouse further to interact with it. Moving the dialog box is a design decision that could be reversed after regression testing the efficiency of the new design reveals this problem.

Looking at the histograms, they provide information never seen before with predictive human performance modeling, i.e., the distribution of times associated with methods using different UI features and methods using a mixture of those features. The histograms for *Insert Hyperlink* and *Insert Table* (right half of Figures 4.4 and 4.5), indicate that the progressive addition of features changes the range and mean of the distribution but not the basic shape. However, the addition of toolbars change the shape of *Format Text* and *Absolute Value* (left half of Figures 4.4 and 4.5) from being clustered around a mean to being bimodal. This can have implications for training, guiding users to the lower cluster of methods, allowing them to be far more efficient with the new version than with the older versions of the system.

Comparing CogTool-Helper's analyses to those used by user experience professionals in the past, consider the different information provided by this extensive exploration of possible methods, compared to modeling individual methods by hand. Typically, the user experience professional would model one method using the menus, one using keyboard shortcuts as much as possible, and one using toolbars as much as possible (e.g., [7]). These models for the *Insert Hyperlink* task predict times of 19.4s, 18.3s and 18.0s, respectively, all in the lower portion of the bimodal distribution. The by-hand analysis still shows that the progressive addition of features is improving efficiency, but does not reveal the possible poor performance cluster of methods that might be avoided through training.

Finally, we note that CogTool was designed to allow UI designers to analyze UI design proposals *before* implementing them. Thus, the analyses shown here, generated from existing systems, could be the baseline for examining new design ideas. If UI designers do not need to spend as much time modeling benchmark tasks on a baseline existing system, they may employ CogTool in the traditional way to assess a broader range of design ideas, avoiding premature focus on a single design direction.

These results arise from using an equal weighting of all test cases to determine values in Table 4.2 and draw the histograms. In the absence of real-world information about our fictitious versions of the system and tasks, we used the assumption of equal weighting to demonstrate some of the practical implications of this work. However this weighting reflects the assumption that end-users in the real world will perform the task using the method in each test case an equal number of times. This is not necessarily a realistic assumption. Card, Moran and Newell [10] observed that people select their methods based on personal preferences (e.g., some prefer menus, others prefer keyboard shortcuts) or characteristics of the task (e.g., at one point in a task the user's hand is already on the mouse, so it is more likely the user will use a toolbar button than if the user's hand is on the keyboard). If the analysis is of a system already released, the user experience professional may have log data to refine the weighting assumption, or prior experience with the user community (similar to Card, Moran and Newell's observations) may influence the weights. The values and histograms will change but the information they provide can be used to reveal properties of the UI design as illustrated above.

CogTool's implementation of the KLM is considered to be +-10% of the average human performance a user experience professional would observe were he or she able to collect skilled time empirically. Part of the variability in human behavior that KLM and CogTool did not capture in previous studies is just what we are exploring here,

i.e., the variation in ways to accomplish a task that skilled users exhibit. Another factor is normal variation in all human performance between and within individuals (e.g., slower performance when fatigued, faster after having drunk a cup of coffee, etc.). HCI research is just beginning to explore modeling tools that predict the latter (e.g., [41]) and ours is the first tool we know of to make it easy to predict the former.

These new tools will allow HCI researchers to understand the contributing factors of variation and how they combine. But until validation research progresses in HCI, it is premature to proclaim that the results in Table 2 and the histograms should be trusted to make important marketing, sales, or UI design decisions, especially because many of the differences are within 10% of each other. That said, it is important to explore the types of information our tool *could* provide, and the types of interpretation a project team *might* make, as the science of predicting variability matures.

Summary of RQ1. This analysis shows that the ability to automate the generation of human performance models for efficiency prediction provides more information than has previously been available to UI designers. This has the potential to provide evidence for marketing and sales campaigns, support the development of appropriate training materials, and feed into the revision of UI designs.

4.4.2 RQ2: Impact of Sampling

With this research question, we seek to examine the impact of the inferred methods discovered by CogTool-Helper. We want to evaluate whether we can sample test cases from the full generated set for a task, providing only a subset to CogTool-Helper and inferring the rest, rather than generating and running every test case for every task. We believe this will help in the scalability of large tasks and when time is limited as

Table 4.3: Results of Sampling Test Cases for Version (MKT) (average of 5 runs)

Design Construction			**CogTool Analysis**				
Task (Sample %/size)	**Run Time(m)**	**%Red**	**No. Methods**	**No. Inferred**	**Mean Time(s)**	**Min Time(s)**	**Max Time(s)**
Format Text (5%/4)	8.5	94.3	12.8	8.8	11.9	9.9	13.4
Format Text (10%/8)	15.3	89.8	41.4	33.4	11.6	8.5	13.7
Format Text (25%/20)	36.8	75.4	76.2	56.2	11.5	8.3	13.7
Format Text (50%/41)	77.4	48.3	81.0	40.0	11.5	8.3	13.7
Format Text (All)	149.8	–	81.0	–	11.5	8.3	13.7
Insert Hyperlink (5%/1)	3.5	90.4	1.0	0.0	19.6	19.6	19.6
Insert Hyperlink (10%/2)	5.6	84.4	3.4	1.4	20.1	19.1	21.0
Insert Hyperlink (25%/5)	12.6	65.0	15.6	10.6	19.7	17.5	21.5
Insert Hyperlink (50%/9)	20.1	44.4	18.0	9.0	19.8	17.5	21.6
Insert Hyperlink (All)	36.1	–	18.0	–	19.8	17.5	21.6
Absolute Value (5%/4)	14.6	93.7	14.8	10.8	17.6	15.2	18.8
Absolute Value (10%/7)	23.7	89.7	25.8	18.8	16.9	14.1	18.7
Absolute Value (25%/18)	59.4	74.3	56.4	38.4	17.0	14.1	18.9
Absolute Value (50%/36)	116.2	49.7	69.6	33.6	17.1	14.1	18.9
Absolute Value Task (All)	231.1	–	72.0	–	17.1	14.1	18.9
Insert Table (5%/2)	5.0	92.2	3.6	1.6	12.3	11.8	12.7
Insert Table (10%/4)	8.7	86.6	6.4	2.4	12.3	11.8	12.8
Insert Table (25%/9)	17.3	73.2	19.4	10.4	12.3	11.4	13.1
Insert Table (50%/18)	34.5	46.5	32.8	14.8	12.4	11.4	13.3
Insert Table Task (All)	64.5	–	36.0	–	12.3	11.3	13.3

is usually the case in regression testing. Running fewer test cases can decrease the run-time of CogTool-Helper considerably, but can decrease the span of performance times the designer has to look at.

Table 4.3 shows the data for each task on the last version (MKT) sampled at 5, 10, 25 and 50% along with the number of test cases (sample size). We show the time in minutes taken, averaged over five samples, for CogTool-Helper to run the test cases and create the designs and tasks, followed by the average percent reduction over running all of the test cases. We list the average number of methods in the resulting CogTool project, along with the average number of inferred methods. The last three columns show the times in seconds of the CogTool human performance predictions (mean, minimum and maximum). The last row of each task contains data for the full set of test cases.

In the 5% sample, we see between a 90.4% and 94.3% reduction in run-time of CogTool-Helper, but we also see a loss in the range of predicted human performance times. In the worst case, (*Insert Hyperlink*), the samples have a single test case with zero inferred methods; the UI designer has only a single point of reference. In the *Absolute Value* task, we lose 1.1 seconds on the minimum predicted performance times, so the UI designer would be missing the most efficient set of methods to perform the task. In the *Insert Table* task, the maximum time gathered is 0.6 seconds less than the maximum time found for all of the test cases. This could cause the designer to miss out on the upper range of the most inefficient methods. Clearly, there would be a tradeoff here in the completeness of the analysis.

The 10% sample shows between 84.4 and 89.8% reduction. We do not lose much on the Min and Max times found, but we still do not find even half of the total number of methods. In the case of *Insert Hyperlink*, we find only 18.8 percent of the total methods. For the 25% samples, we get much closer to obtaining the total number of

test cases but still miss out on 15.6 methods (average) in *Absolute Value*, and 16.6 methods in the *Insert Table* task. Examination of the detail in Table 4.3 shows that in the 25% samples, the inferred methods algorithm created more than 50% and as high as 94% of the complete set of methods across the four tasks, while the average time to construct the designs is approximately one fourth of the time taken for the full set.

In all four tasks, the 50% samples have the full range of human performance values, and we either generate all of the methods that are possible for that task with the inferred method algorithm (*Format Text* and *Insert Hyperlink*), or come within 10% of all possible methods (*Absolute Value* and *Insert Table*). The runtime savings are over 50%, which equates to almost 2 hours in the *Absolute Value* task.

To understand the tradeoffs of the mid-level sampling (10 - 25%), we built histograms for each individual sample and compared this with the original histograms. Consider the *Insert Hyperlink* task in Figure 4.6. The 10% sample is shown in the top row and the 25% sample in the bottom row. In the 10% sample, we do not consistently see the full range of human performance values (e.g. samples 3 and 4). However, at the 25% sampling level the graphs appear to have a similar range to our original histograms (right side of Figure 4.4). We see different results for the *Format Text* task (Figure 4.7). Both the 10% and 25% samples show a very similar shape to those in the left side of Figure 4.4. Looking at the *Absolute Value* and *Insert Table* task (Figure 4.8 and Figure 4.9), we noticeably miss out on information in the 10% samples, but this improves when moving to 25%. Clearly, the effectiveness of the sampling depends on the properties of the task we are sampling test cases from and the interface on which it is performed.

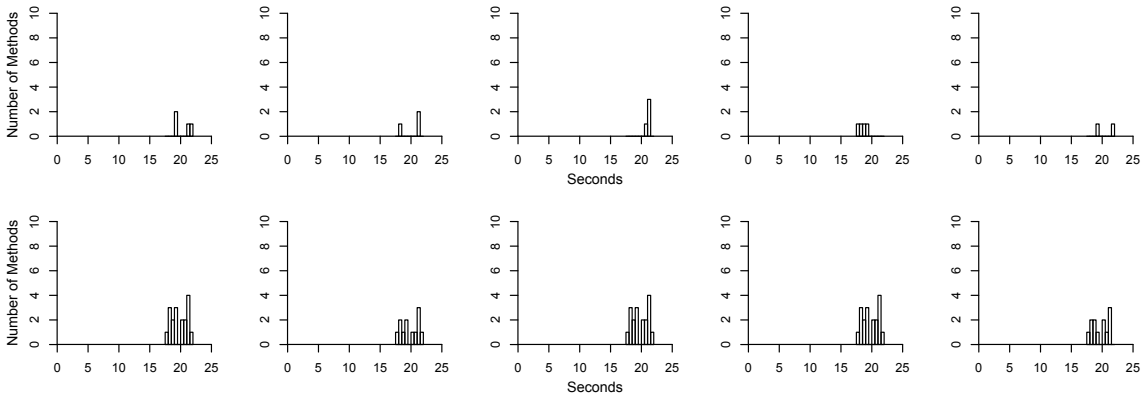


Figure 4.6: Insert Hyperlink Task (MKT) Sampled at 10 (top) and 25 (bottom) percent

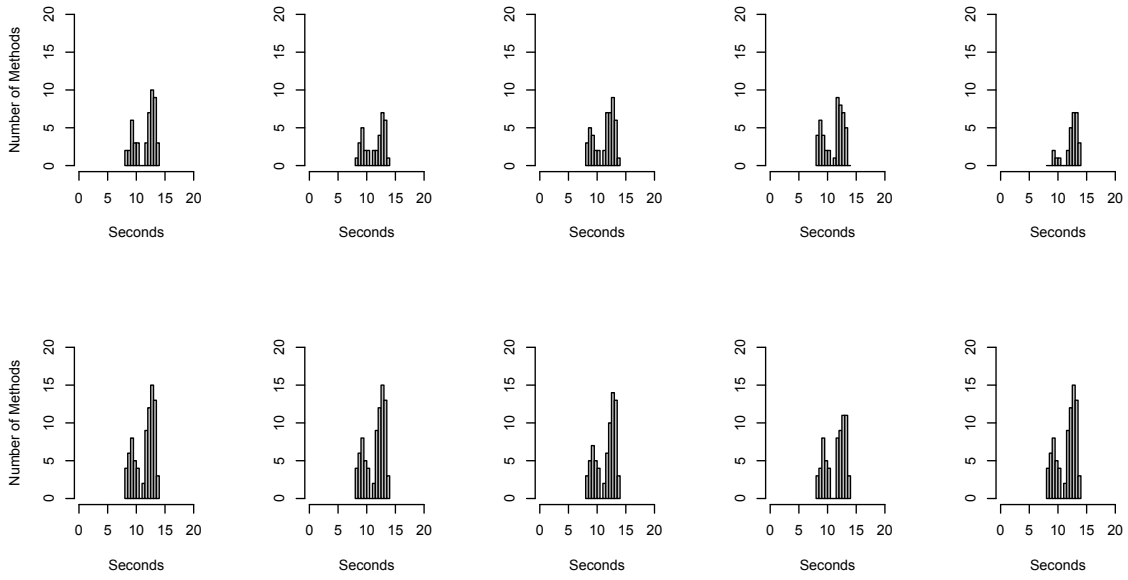


Figure 4.7: Format Text Task (MKT) Sampled at 10 (top) and 25 (bottom) percent

Summary of RQ2. We have shown that sampling test cases from the full set of generated test cases for a task can increase the efficiency of design construction, while inferred methods allow us to retain much of the information. At sampling levels of

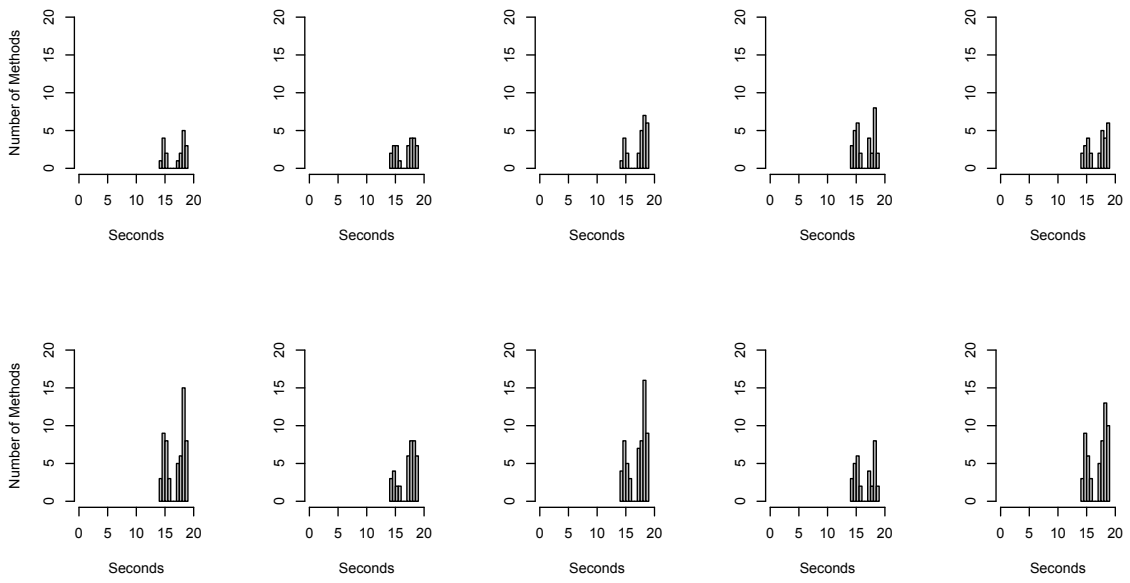


Figure 4.8: Absolute Value Task (MKT) Sampled at 10 (top) and 25 (bottom) percent

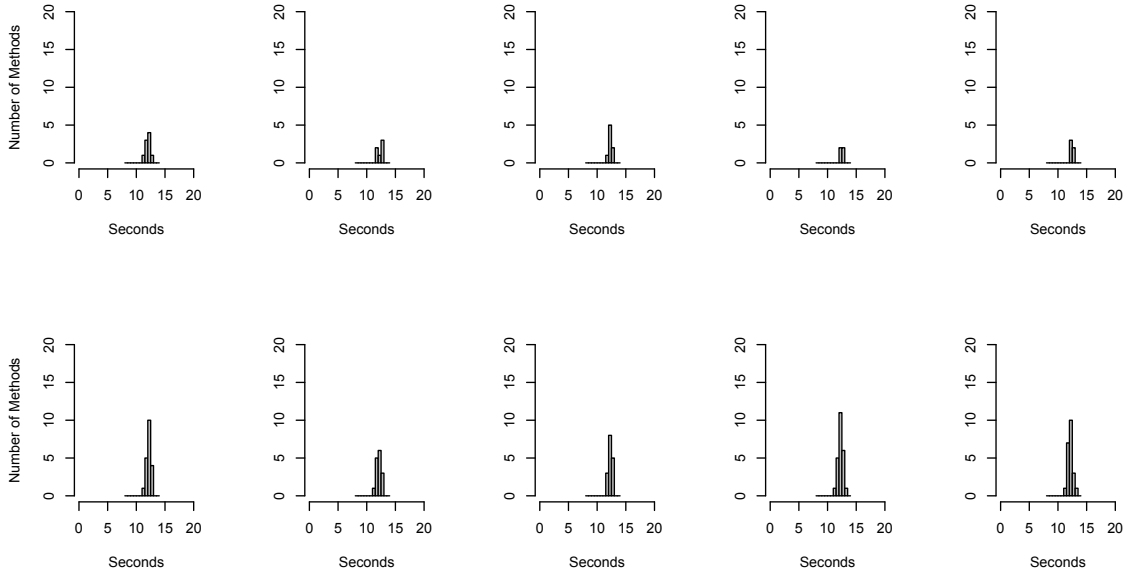


Figure 4.9: Insert Table Task (MKT) Sampled at 10 (top) and 25 (bottom) percent

25 or 50%, the UI designer still has enough methods to reason about the benefits or detriments of their new design and can save time spent on constructing CogTool designs.

4.4.3 Summary

In this chapter, we presented a method for generating meaningful test cases for a task. We showed that these test cases can provide very valuable information to UI designers and regression testers. We also conducted a set of sampling experiments to evaluate the power of CogTool-Helper's inferred methods algorithm and showed that even with a 50% reduction in the number of test cases we provide to CogTool-Helper, we can provide the same amount of information to the UI analyst in half the time.

Chapter 5

Conclusions and Future Work

In this thesis, we presented CogTool-Helper, a tool that extends CogTool by providing an automated way to generate design storyboards, tasks, and methods for a legacy application, as well as a way to uncover implicit methods that exist on a design for a task. This allows the design analyst to spend their time actually analyzing their designs instead of taking screenshots, adding widget overlays, drawing transitions, and demonstrating tasks. By inferring additional methods, this may allow the analyst to discover alternate ways of performing their task that they may not have been aware of but might be important to consider. These methods might turn out to be the most efficient way of performing the task.

Additionally, we have developed a method for utilizing functional GUI testing tools to generate meaningful methods for tasks to help UI analysts evaluate all of the possible ways a user could perform a task on their interface; a step towards automated regression testing for human performance. We performed a feasibility study using three modules of LibreOffice and found that the information can provide a rich set of data. This may allow project teams to explore the impact of changes to their design more thoroughly, and allow them to use currently existing testing strategies to be

extended to test for user performance. With this study, we also explored the impact of CogTool-Helper's inferred method algorithm on optimizing the process. In all four of the tasks we examined, we were able to provide the same amount of information when sampling only fifty percent of the test cases, thereby cutting the runtime of design construction significantly. Referring back to our UI designer, Sally, this would cut her time to build her UI models and tasks significantly and allow her to explore all of the possible ways her task could be performed in her interface, providing her with more information, and allowing her to spend more quality time on her analysis.

There are several areas for future work we would like to explore with CogTool-Helper. First, we would like to expand CogTool-Helper to support more application types, including all of those types supported by GUITAR. So far, we have focused only on using testing tools to facilitate creation of CogTool models of legacy applications, this means that we need the GUI and application functionality to be implemented. However, modeling in CogTool is often used in design exploration, allowing detection of usability issues prior to coding. Facilitating the creation of such models may be another area where GUI testing technologies might be able to help. In this case, testing technologies such as [11], which uses a computer vision strategy, might facilitate the creation of human performance models from a user interface sketch. Additionally, we would like to see if we can support model creation for tasks involving the use of multiple applications. This is a concern critical to HCI but too often ignored in software testing, even though these interactions are the source of many errors.

Experimentation with our extension to CogTool-Helper for test generation has led to some interesting future directions. First, we believe that we have identified a good starting set of rules, but there may be additional rules required for more complex tasks. For instance, our current task specific rule for repeat is binary, meaning an event can appear either once or more than once, but we anticipate that there may

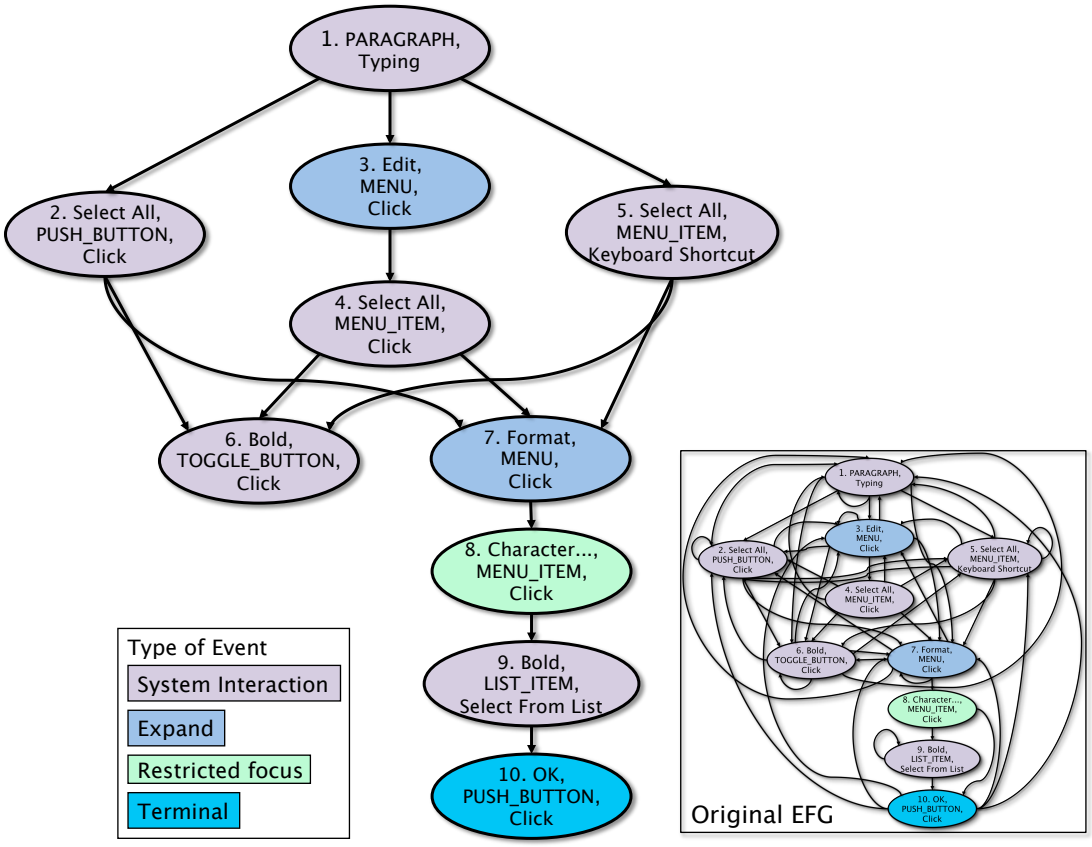


Figure 5.1: Reduced EFG

be a need to provide a finer granularity and to specify a range of cardinalities. The improvement that we believe can have the greatest impact on scalability is to the implementation of our test case generation (Step D in Figure 4.1). For this study we use the existing GUITAR test case generator which traverses all paths of the given length on the EFG and apply rules on each resulting test case. Since the number of paths grows combinatorially, this will not scale to large tasks. However, if we apply the rules first (or as we explore the graph), then we can avoid this bottleneck. The reduced EFG (Figure 5.1) for this task has the same number of nodes, but only 13 edges (a 75% reduction). We plan to modify our process, by either performing a

transformation of the EFG based on the rules first, or through dynamic pruning of the graph as we generate tests.

Finally, our current process requires the complementary skills of both a testing professional and a user experience professional. For example, some of the information necessary to specify the event tuples is not easily obtained by looking at the interface, but must be extracted through iteration with the ripper, often not accessible to UI designers. This suggests a need for a capture mechanism to obtain the tuples automatically. In order for our process to be adopted into mainstream regression testing, work needs to be done to make the tools usable by the right person or people in a software project team.

Appendix A

Widget and Action Glossary

A.1 CogTool Widget Type Translations

Button

UNO Widget Class: PUSH_BUTTON, **JFC Widget Class:** javax.swing.JButton

How it is represented: The property `Can be toggled` is not checked.

```
<widget w-is-selected="false" w-is-standard="true"
   name="WidgetName" shape="rectangle"
   w-is-toggleable="false" type="button">
   <displayLabel>WidgetLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Check Box

UNO Widget Class: CHECK_BOX, **JFC Widget Class:** javax.swing.JCheckBox

Widget Properties: Each check box has the CogTool widget type `Check Box`. For checkboxes that are initially selected, the checkbox `Initially Selected` in CogTool is checked. This corresponds to `w-is-selected` in the XML. Checkboxes that belong

in a group should be represented as one by giving them the sample group `name` in the XML representation. The sample below contains a check box group with two checkboxes.

```
<widget w-is-selected="true" w-is-standard="true"
   name="Option␣1" shape="rectangle"
   type="check␣box" x="0" y="0" group="CheckboxGroup1">
   <displayLabel>Option 1</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
<widget w-is-selected="false" w-is-standard="true"
   name="Option␣2" shape="rectangle"
   type="check␣box" x="0" y="0" group="CheckboxGroup1">
   <displayLabel>Option 1</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Link

UNO Widget Class: HYPER_LINK, **JFC Widget Class:** javax.swing.JTextArea

Widget Properties: Links are represented by setting the type attribute to `link`.

```
<widget w-is-standard="true" name="Google" shape="rectangle"
   type="link">
   <displayLabel>www.google.com</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

List Box Item

UNO Widget Class: LIST_ITEM, **JFC Widget Class:** javax.swing.JList

Widget Properties: A widget with the class `LIST_ITEM` is a list box item if the immediate parent of the parent `LIST` object does not have the type `COMBO_BOX`. For JFC widgets, all of the items in the list (options in javax.swing.JList) should be

represented as a list box item. All items in the list box should be grouped. They should have same value for the `group` attribute.

```
<widget w-is-standard="true" name="ListBoxItemName"
   group="ListBoxGroup1" shape="rectangle"
   type="list box item">
   <displayLabel>ListBoxItemLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Menu

UNO Widget Class: MENU, **JFC Widget Class:** javax.swing.JMenu

Widget Properties: A `MENU` should be represented as a `Menu` widget type if it is a top-level menu. Otherwise, it should be represented as a `Submenu`. A menu can have children of the type `Menu Item` and `Submenu` (which can have children themselves). All menus belonging to the same menu bar should be in a group. They should have the same value for the `group` attribute.

```
<widget w-is-standard="true" name="MenuName"
   group="MenuGroup1" shape="rectangle" type="menu">
   <displayLabel>MenuLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Menu Item

UNO Widget Class: MENU_ITEM, **JFC Widget Class:** javax.swing.JMenuItem

Widget Properties: The `parent` attribute of the menu item should be set to the name of the immediate parent menu, which can be a submenu or a menu.

```
<widget w-is-standard="true" name="MenuItemName"
   shape="rectangle" type="menu item"
```

```
    parent="MenuItemParentName">
    <displayLabel>MenuItemLabel</displayLabel>
    <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Menu Bar

UNO Widget Class: MENU_BAR, **JFC Widget Class:** javax.swing.JMenuBar

Widget Properties: A menu bar allows menus to be grouped. Every menu that is located on the same menu bar is in the same group. All of these menus are given the same `group` attribute value.

Menu Separator

UNO Widget Class: SEPARATOR, **JFC Widget Class:** javax.swing.JSeparator

Widget Properties: A separator is represented in the same way as a menu item. It also has the additional attribute `w-is-separator` set to *true*, which corresponds the the `Separator` option being checked in CogTool. It should also have the `parent` attribute set to the name of the immediate parent menu (one level up).

```
<widget w-is-separator="true" w-is-standard="true"
    name="SeparatorName" shape="rectangle" type="menu item"
    parent="SeparatorParentName">
    <displayLabel>SeparatorLabel</displayLabel>
    <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Popup Menu

UNO Widget Class: POPUP_MENU,

JFC Widget Class: javax.swing.JPopupMenu

Not yet supported

Pull-Down List

UNO Widget Class: LIST, **JFC Widget Class:** javax.swing.JComboBox

Widget Properties: A pull-down list is a widget that opens up a list of options that the user can select from. The `LIST` Class should be represented as a pull-down list if the `Class` of its immediate parent is `COMBO_BOX`. The `javax.swing.JComboBox` Class should be represented as a pull-down list directly with each of its list items represented as a `pull-down list item`.

```
<widget w-is-standard="true" name="PullDownListName"
   shape="rectangle" type="pull-down list">
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Pull-Down List Item

UNO Widget Class: LIST_ITEM, **JFC Widget Class:** javax.swing.JComboBox

Widget Properties: A `LIST_ITEM` Class widget should be represented as a pull-down list item if the parent of its immediate parent widget has the class `COMBO_BOX`. Otherwise, the item should be represented as a list box item.

```
<widget w-is-standard="true" name="PullDownItemName"
   shape="rectangle" type="pull-down item"
   parent="PullDownListName">
   <displayLabel>PullDownItemLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Radio Button

UNO Widget Class: RADIO_BUTTON, **JFC Widget Class:** javax.swing.JRadioButton

Widget Properties: A radio button has the `type` set to *radio button*. The radio

button that is selected in the group has the property `Initally selected` checked, so the attribute `w-is-selected` should be included and have the value *true*. Radio buttons should be grouped according to their radio button groups. They should have the samve value for the `group` attribute.

```
<widget w-is-selected="true" w-is-standard="true" name="Yes"
   shape="rectangle" type="radio button"
   group="RadioButtonGroup1">
   <displayLabel>Yes</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
<widget w-is-standard="true" name="No" shape="rectangle"
    type="radio button" group="RadioButtonGroup1">
   <displayLabel>No</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Submenu

UNO Widget Class: MENU, **JFC Widget Class:** javax.swing.JMenu

Widget Properties: A `MENU` should be represented as a Submenu if it is not a top-level menu. It should have the parent attribute value of the name of the immediate parent menu (one level up). It should also have the `type` attribute set to *submenu*.

```
<widget w-is-standard="true" name="SubmenuName"
   shape="rectangle" type="submenu" parent="ParentMenuName">
   <displayLabel>SubmenuLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Tab

UNO Widget Class: PAGE_TAB, **JFC Widget Class:** javax.swing.JTabbedPane

Widget Properties: Each tab is represented as a `Button` type. Tabs are represented

like toggle buttons. The property `Can be toggled` (w-is-toggleable) is checked, and if the tab is currently selected, the property `Initally selected` (w-is-selected) is also checked. The JFC widget Class `javax.swing.jTabbedPane` contains widgets for each tab so each of these should be represented as a separate button.

```
<widget w-is-selected="true" w-is-standard="true"
   name="WidgetName" shape="rectangle"
   w-is-toggleable="true" type="button">
   <displayLabel>WidgetLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Text

UNO Widget Class: PARAGRAPH, TEXT, LABEL,

JFC Widget Class: javax.swing.JTextBox, javax.swing.JTextArea, javax.swing.JLabel

Widget Properties: Any text content of a text box or label should be represented as a `Text` widget. For the `LABEL` type, set the `displayLabel` property in the XML to the text of the label directly (this is usually the GUITAR `Title` property). For the `PARAGRAPH` and `TEXT` widget types, the text content is extracted from an editable text object, and a widget is built just representing the text in the text box only, as the `PARAGRAPH` and `TEXT` objects are represented as text boxes separately.

```
<widget w-is-standard="true" name="TextName"
   shape="rectangle" type="text">
   <displayLabel>TextContent</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Text Box

UNO Widget Class: PARAGRAPH, TEXT,

JFC Widget Class: javax.swing.JTextBox, javax.swing.JTextArea

Widget Properties: A `text box` widget is created for each `PARAGRAPH` and `TEXT` widget. Also, a `text` widget is created for any text content in the `PARAGRAPH` and `TEXT` widgets. The same strategy is followed for `javax.swing.JTextBox` and `javax.swing.JTextArea`.

```
<widget w-is-standard="true" name="TextBoxName"
   shape="rectangle" type="text box">
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

Toggle Button

UNO Widget Class: TOGGLE_BUTTON,

JFC Widget Class: javax.swing.JToggleButton

Widget Properties: The property `Can be toggled` is checked.

```
<widget w-is-selected="false" w-is-standard="true"
   name="WidgetName" shape="rectangle"
   w-is-toggleable="true" type="button">
   <displayLabel>WidgetLabel</displayLabel>
   <extent height="--" y="--" width="--" x="--"/>
</widget>
```

A.2 CogTool Action Translations

CogTool-Helper can translate 6 action types from GUITAR actions into CogTool transitions. All are supported for OpenOffice interfaces, and most are supported for Java interfaces. The action types and their associated CogTool transitions are outlined in Table A.2 and defined in further detail in the following sections. .

Action	Description	CogTool Transition Type
Left-Click	Performs a left click on the widget	Mouse Transition
Select From List	Selects an item from a list	Mouse Transition
Set Value	Set the value of a table cell	Keyboard Transition
Type	Performs a typing action	Keyboard Transition
Keyboard Shortcut	Performs a keyboard shortcut	Keyboard Transition
Keyboard Access	Performs a keyboard access for hierarchical menus	Keyboard Transition

Table A.1: Actions available in CogTool-Helper

A.2.1 Mouse Transitions

In CogTool, a Mouse Transition has the properties `Mouse Button`, `Action`, `Source`, `Destination`, `Modifiers`, and `Wait for system response`. The `Mouse Button` property can take 3 values: *Left*, *Middle*, and *Right*. These represent the three buttons on a typical mouse. The `Action` property can take 6 values: *Click*, *Double-Click*, *Triple-Click*, *Press*, *Release*, and *Hover*. The `Source` property represents the source frame of the transition and the `Destination` property represents the destination of the transition. The `Modifiers` property represents the keyboard modifiers that can be added to the mouse actions. We do not yet support any modifiers in CogTool-Helper. The last property `Wait for system response` represents the delay that the system would have after performing the action before its result can be seen. We do not support `Wait for system response` in CogTool-Helper, so by default this property is set to 0.0 seconds.

This XML snippet shows how a transition for a mouse action is represented in CogTool XML. The `durationInSecs` attribute corresponds to the CogTool property, `Wait for system response`. The `destinationFrameName` attribute is the name of the frame the transition is pointing to and sets the `Destination` property in CogTool.

The `Target` for the transition is the name of widget where the transition originates, and corresponds to the `Source` property in CogTool. A transition has an `Action` property in CogTool which corresponds to `action` in the XML representation. In this case, it is of the `mouseAction` type. Each `mouseAction` has a `button` and an `action` which define the `Mouse Button` and `Action` properties for the transition.

```
<widget name="....">
   <transition durationInSecs="..."
   destinationFrameName="....">
      <action>
         <mouseAction button="...." action="....">
         </mouseAction>
      </action>
   </transition>
</widget>
```

A.2.1.1 Left-Click

The Left-Click action performs a left click on a widget. In GUITAR, this maps to the OpenOffice action handler *OOActionHandler*. It also maps to the action handler *JFCActionHandler* for Java interfaces. The Left-Click action sets the `Mouse Button` property to the value *Left* and the `Action` property to the value *Click*. `Wait for system response` is set to the value *0.0 seconds*.

```
<transition durationInSecs="0.0" destinationFrameName="....">
   <action>
      <mouseAction button="left" action="downUp">
      </mouseAction>
   </action>
</transition>
```

In the above XML representation, the `button` attribute is *left* corresponding the `Left` Mouse Button property in CogTool. The `Click` action is represented as `downUp`.

A.2.1.2 Select From List

The Select From List action selects an option from a list. This can be a combo box (where the list is opened to select from options), or a list-box where all list items are displayed and one is clicked directly to select it from the list. Select From List is represented as a Mouse action in CogTool. In GUITAR, Select From List maps to the OpenOffice action handler *OOSelectFromParentHandler* and in Java maps to the hander *JFCSelectFromParentHandler*. The Select From List action sets the `Mouse Button` property in CogTool to *Left* and the `Action` property to *Click*. `Wait for system response` is set to *0.0 seconds*. The target for this transition is CogTool widget for the option in the combo box or the list box item that is being selected.

The CogTool XML representation for the Select From List transition is encoded as follows. The `button` attribute is *left* corresponding the `Left` Mouse Button property. The `Click` Action is represented as *downUp*.

```
<transition durationInSecs="0.0" destinationFrameName=
      "....">
   <action>
      <mouseAction button="left" action="downUp">
      </mouseAction>
   </action>
</transition>
```

A.2.2 Keyboard Transitions

A keyboard transition in CogTool has 4 properties. These are `Text`, `Source`, `IsCommand`, and `Wait for system response`. `Text` is the text that is being typed in. There is no limit to the amount of text than can be typed in for a keyboard transition. `Source` is the name of the source frame of the keyboard transition. Notice that this is not

the same as a mouse transition whose source is the widget being acted upon. The property `IsCommand` tells CogTool whether the keyboard transition is a keyboard command sequence. If set to false, the text is just interpreted as a regular string of typed text. The property `Wait for system response` is not supported though CogTool-Helper, but is always set to *0.0 seconds* for every keyboard transition.

The XML for the keyboard transition is represented as follows. The source frame of this keyboard transition contains the XML representation. A Keyboard Transition is represented by the `keyboardTransitions` tag. This tag contains a `transition` tag. The transition has a `destinationFrameName` and a `durationInSecs`, which correspond to the `Destination` and `Wait for system response` properties in CogTool. The transition tag contains a `keyboardAction` tag meaning it is a keyboard transition. It has two properties, *is-command* and `press`. `is-command` corresponds the `IsCommand` property in CogTool. The `keyboardAction` tag contains a `text` tag. This is the actual text representation of what is being typed in. This can be characters for regular text and special characters to represent command sequences, and this tag sets the `Text` property for the keyboard transition in CogTool.

```
<keyboardTransitions>
   <transition destinationFrameName="...."
   durationInSecs="0.0">
      <action>
         <keyboardAction is-command="false" type="press">
            <text><![CDATA[....]]></text>
         </keyboardAction>
      </action>
   </transition>
</keyboardTransitions>
```

There are four different actions supported by CogTool-Helper that correspond to keyboard transitions in CogTool. These are Keyboard Shortcut, Keyboard Access, and Set Value.

A.2.2.1 Keyboard Shortcut

A keyboard shortcut is a sequence of commands executed using the keyboard such as the common 'Ctrl-C' action for copy. Keyboard shortcuts are typically associated with menu items to be a shortcut for that menu action, but can also be associated with other widgets or actions depending on the interface. CogTool-Helper can currently only extract Keyboard Shortcuts for menus and menu items in OpenOffice. The handler representing a keyboard shortcut in OpenOffice is OOKeyboardHandler. There is no supported handler yet for Java interfaces.

In CogTool, the Keyboard Shortcut action sets `IsCommand` property to true. The `Text` property is a string made up of the special characters representing the key commands (e.g., Ctrl, Alt) and the additional characters pressed during the command. A keyboard shortcut also needs a `Destination` and `Wait for system response`.

```
<keyboardTransitions>
   <transition destinationFrameName="...."
   durationInSecs="0.0">
      <action>
         <keyboardAction is-command="true" type="press">
            <text><![CDATA[+c]]></text>
         </keyboardAction>
      </action>
   </transition>
</keyboardTransitions>
```

An example XML representation for the keyboard shortcut action for 'Ctrl-C' is shown above. `is-command` is set to *true*, and the `text` tag contains the special string representing 'Ctrl-C' in CogTool.

A.2.2.2 Keyboard Access

The Keyboard Access action represents the use of keyboard commands plus extra characters for opening up menus and submenus. This is typically available only on Windows systems. OpenOffice only supports this in Windows. In all of the OpenOffice modules, menus can be opened using the 'Alt' command combined with a special character that is underlined on the menu label. Each successive menu item can be performed by typing the underlined letter on the menu item label. In CogTool, these are represented the same way as keyboard shortcuts. The `IsCommand` property is set to *true* and the `Text` for the command is the modifier (Alt) concatenated with each of the characters in succession.

A.2.2.3 Set Value

Set value is used for setting the value of a a table cell in OpenOffice. Set Value corresponds to the action handler *OOValueHandler*. Set Value is represented as a keyboard transition. For Set Value, the property `IsCommand` is set to *false*. The `Text` property is the numbers or text to type into the cell.

The XML representation for the Set Value action is below. `is-command` is *true*. *value* represents the value to be typed in, which can be characters or digits.

```
<keyboardTransitions>
   <transition destinationFrameName="...."
   durationInSecs="0.0">
      <action>
         <keyboardAction is-command="true" type="press">
```

```
            <text><![CDATA[value]]></text>
          </keyboardAction>
        </action>
      </transition>
    </keyboardTransitions>
```

A.2.2.4 Type

The `Type` action can be represented as either a Mouse or Keyboard transition based on the given parameters in the GUITAR test case. There are 5 possible commands for the Type action. These commands are `TextInsert`, `TextReplace`, `Select`, `Unselect`, and `Cursor`. Some of these also have additional parameters. The parameter given in the test case takes on the form of <*Command*>_<*Parameter1*>_<*Parameter2*>, etc. The `TextInsert` command takes two additional parameters; the text to insert <*Text*> and the index in the text to start inserting <*Index*>. `TextReplace` takes on one additional parameter, <*Text*>, the text to replace all of the text in the text box with. `Select` takes two additional parameters; the index to start the selection from <*Index*>, and the number of characters to select. `Unselect` does not take any additional parameters. It unselects the entire block of text. `Cursor` takes one additional parameter; the index to move the cursor in the text <*Index*>. Each of these `Type` commands maps to the following CogTool transition type.

TextInsert maps to a **Keyboard Transition** in CogTool. The `Text` property is set to the value of the <**Text**> parameter. `IsCommand` is *false*.

TextReplace maps to a **Keyboard Transition** in CogTool. The `Text` property is set to the value of <**Text**> parameter. `IsCommand` is *false*.

Select maps to a Mouse Transition in CogTool. CogTool cannot easily represent a "drag" action so we set `Mouse Button` to *Left* and `Action` to *Double-Click* if one word is selected and *Triple-Click* if the entire text is selected.

Unselect maps to a Mouse Transition. `Mouse Button` is set to *Left* and `Action` is set to *Click*

Cursor maps to a Mouse Transition. `Mouse Button` is set to *Left* and `Action` is set to *Click*

Appendix B

Detailed Description of Tasks

The following are detailed descriptions of the tasks we used in our case study covered in Chapter 4. Two tasks utilize the LibreOffice Writer module. The Writer module is a word processing tool for editing documents. Another task uses the Calc module, which is a spreadsheet calculation tool. The last task is in the Impress module which is used for creating presentations. For each task, a high-level description is given for the task, including the goals and approaches defined for achieving that task, a table of the event tuples we defined for the task, and a table summarizing the rules we defined to guide the generation for this task.

B.1 Task 1: Format Text

This task utilizes the LibreOffice Writer interface. It types the word "Google" into the Writer document, selects all of the text, aligns the text to the center, and changes the font weight to bold. The result of performing this task is shown in Figure B.1. The goals and approaches for this task are detailed in Table B.1. The goals do not have to appear in the given order. The event tuples we defined for this task are shown

in Table B.2. The task *Format Text* has 16 possible events. The minimum number of events to achieve the task is 4 and the maximum number of events is 10, so we generated all test cases for each of these lengths. The task has 9 system interaction events, 3 expand events, 2 restricted focus events, and 1 terminal event. Rules for generation are shown in Table B.3. A sample test case for this task is:

1. *PARAGRAPH, Type 'Google', LibreOffice Writer*
2. *Select All, PUSH_BUTTON, Click, LibreOffice Writer*
3. *Format, MENU, Click, LibreOffice Writer*
4. *Character...,MENU_ITEM, Click, LibreOffice Writer*
5. *Bold, LIST_ITEM, Click, Character*
6. *OK, PUSH_BUTTON, Click, Character*
7. *Centered, TOGGLE_BUTTON, Click, LibreOffice Writer*

B.2 Task 2: Insert Hyperlink

This task also utilizes the LibreOffice Writer interface. The purpose of the task is to insert a hyperlink into the document for 'www.amazon.com'. The label text of the hyperlink should be 'Amazon' and the letters should all be in uppercase. The result of performing this task is shown in Figure B.2. Table B.4 summarizes the goals and approaches for this task. There are three ways to open the hyperlink window and three ways to make the text uppercase. Table B.5 lists the event tuples created for this task, color-coded by their event type. For *Insert Hyperlink*, there are 13 events in total; 6 system interaction, 3 expand, 3 unrestricted focus (because the Hyperlink window is non-modal), and 1 terminal. The minimum number of events to accomplish this task is 6 and the maximum number of events is 9. We generated test cases for lengths 6, 7, 8, and 9. Table B.6 summarizes the rules applied during generation for this task. A sample generated test case is:

1. *Hyperlink, MENU_ITEM, Keyboard Access, LibreOffice Writer*
2. *Target, TEXT, Type 'www.amazon.com', Hyperlink*
3. *Text, TEXT, Type 'Amazon', Hyperlink*
4. *Apply, PUSH_BUTTON, Click, Hyperlink*
5. *Close, PUSH_BUTTON, Click, Hyperlink*
6. *UPPERCASE, PUSH_BUTTON, Click, LibreOffice Writer*

B.3 Task 3: Absolute Value

The purpose of this task is to insert an absolute value function for the number '-87' in Cell A1. The cell should be shifted to the right by one cell, and the 'Column & Row Headers' should be turned off. The result of performing this task is shown in Figure B.3. The goals and approaches for this task are shown in Table B.7. There are 3 ways to insert a function, 3 ways to shift cells right, and 2 ways to turn of Column & Row Headers. The task consists of 16 events and has a minimum and maximum length of 7 and 12, respectively. The events making up this task are listed in Table B.8. Rules for *Absolute Value* are shown in Table B.9. Using these rules applied to the generation resulted in a total of 72 test cases. A sample generated test case that performs this task is:

1. *Cell A1, TABLE_CELL, Click, LibreOffice Calc*
2. *Function Wizard, PUSH_BUTTON, Click, LibreOffice Calc*
3. *Next, PUSH_BUTTON, Click, Function Wizard*
4. *Number, TEXT, Type '-87', Function Wizard*
5. *OK, PUSH_BUTTON, Click, Function Wizard*
6. *View, MENU, Click, LibreOffice Calc*
7. *Column & Row Headers, Click, LibreOffice Calc*
8. *Insert Cells Right, PUSH_BUTTON, Click, LibreOffice Calc*

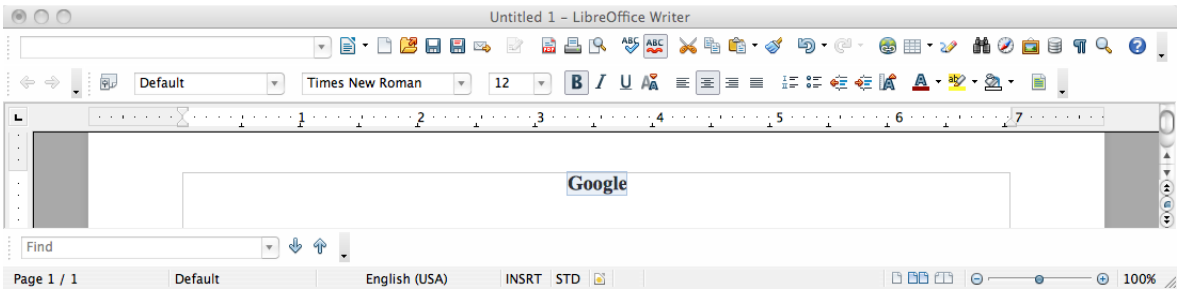


Figure B.1: Result of performing Format Text task

B.4 Task 4: Insert Table

The fourth task utilizes the LibreOffice Impress interface. The task performs 3 separate actions, inserting a new slide, inserting a 6 column, 5 row table on the first slide, and hiding the task pane. The result of this task is shown in Figure B.4. The goals and approaches for this task are shown in Table B.10. There are 2 ways to open the Table window, 3 ways to add a new slide, and 2 ways to turn off the task pane.The event tuples we defined for this task are shown in Table B.11. There are 11 events in total: 6 system interaction, 2 expand, 2 restricted focus, and 1 terminal. We applied the rules in Table B.12 to generate test cases for *Insert Table*. This resulted in a total of 36 total test cases. A sample generated test case to perform this task is:

1. *Table, MENU_ITEM, Keyboard Access, LibreOffice Impress*
2. *Columns, TEXT, Type '6', Insert Table*
3. *OK, PUSH_BUTTON, Click, Insert Table*
4. *Slide, PUSH_BUTTON, Click, LibreOffice Impress*
5. *View, MENU, Click, LibreOffice Impress*
6. *Task Pane, MENU_ITEM, Click, LibreOffice Impress*

Table B.1: Goals and approaches f or Format Text task

Goal	Approaches	Methods
Type Text: “Google”		
Select all of the text.	(a) Menu (b) Keyboard (c) Toolbar	(a) Edit→Select All (b) Type ‘Ctrl-A’ (c) Select All
Change font weight to Bold.	(a) Menu (b) Keyboard (c) Toolbar	(a) Format→Character..., Bold, OK (b) Type ‘Alt-OH’, Bold, OK (c) Bold
Align text to center.	(a) Menu (b) Keyboard (c) Toolbar	(a) Format→Alignment→ Centered (b) Type ‘Alt-OTC’ (c) Centered

Table B.2: Event tuples for Format Text task

Title	Class	Action	Window
	PARAGRAPH	Type, ‘Google’	LibreOffice Writer
Select All	PUSH_BUTTON	Click	LibreOffice Writer
Select All	MENU_ITEM	Click	LibreOffice Writer
Select All	MENU_ITEM	Keyboard Shortcut	LibreOffice Writer
Bold	TOGGLE_BUTTON	Click	LibreOffice Writer
Centered	MENU_ITEM	Keyboard Access	LibreOffice Writer
Centered	MENU_ITEM	Click	LibreOffice Writer
Centered	TOGGLE_BUTTON	Click	LibreOffice Writer
Bold	LIST_BOX_ITEM	Click	Character
Format	MENU	Click	LibreOffice Writer
Edit	MENU	Click	LibreOffice Writer
Alignment	MENU_ITEM	Click	LibreOffice Writer
Character...	MENU_ITEM	Click	LibreOffice Writer
Character...	MENU_ITEM	Keyboard Access	LibreOffice Writer
OK	PUSH_BUTTON	Click	Character
System Interaction	Expand	Restricted Focus	Terminal

Table B.3: Rules for Format Text task

Rule	Events
Exclusion 1	Select All, PUSH_BUTTON, Click, LibreOffice Writer Select All, MENU_ITEM, Click, LibreOffice Writer Select All, MENU_ITEM, Keyboard Shortcut, LibreOffice Writer
Exclusion 2	Centered, MENU_ITEM, Keyboard Access, LibreOffice Writer Centered, MENU_ITEM, Click, LibreOffice Writer Centered, TOGGLE_BUTTON, Click, LibreOffice Writer
Exclusion 3	Bold, TOGGLE_BUTTON, Click, LibreOffice Writer Bold, LIST_BOX_ITEM, Click, Character
Order 1	PARAGRAPH, Type, 'Google', LibreOffice Writer
	Select All, PUSH_BUTTON, Click, LibreOffice Writer Select All, MENU_ITEM, Click, LibreOffice Writer Select All, MENU_ITEM, Keyboard Shortcut, LibreOffice Writer Centered, MENU_ITEM, Keyboard Access, LibreOffice Writer Centered, MENU_ITEM, Click, LibreOffice Writer Centered, TOGGLE_BUTTON, Click, LibreOffice Writer Bold, TOGGLE_BUTTON, Click, LibreOffice Writer Bold, LIST_BOX_ITEM, Click, Character
Order 2	Select All, PUSH_BUTTON, Click, LibreOffice Writer Select All, MENU_ITEM, Click, LibreOffice Writer Select All, MENU_ITEM, Keyboard Shortcut, LibreOffice Writer
	Bold, TOGGLE_BUTTON, Click, LibreOffice Writer Bold, LIST_BOX_ITEM, Click, Character
Required	PARAGRAPH, Type, 'Google', LibreOffice Writer
Repeat	Format, MENU, Click, LibreOffice Writer

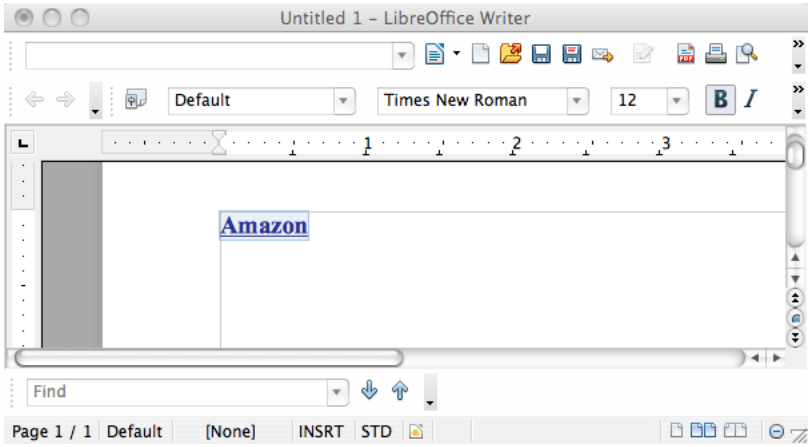


Figure B.2: Result of performing Insert Hyperlink task

Table B.4: Goals, Approaches, and Methods for Insert Hyperlink Task

Goal	Approaches	Methods
Open the 'Hyperlink' window.	(a) Menu (b) Keyboard (c) Toolbar	(a) Insert→Hyperlink (b) Type 'Alt-IH' (c) Hyperlink
Type 'www.amazon.com' as Target of hyperlink.		
Type 'Amazon' as Text of hyperlink.		
Click 'Apply' button.		
Click 'OK' button.		
Make font of hyperlink uppercase.	(a) Menu (b) Keyboard (c) Toolbar	(a) Format→Change Case→UPPERCASE (b) Type 'Alt-OCU' (c) UPPERCASE

Table B.5: Event tuples for Insert Hyperlink task

Title	Class	Action	Window
UPPERCASE	MENU_ITEM	Click	LibreOffice Writer
UPPERCASE	MENU_ITEM	Keyboard Access	LibreOffice Writer
UPPERCASE	PUSH_BUTTON	Click	LibreOffice Writer
Target	TEXT	Type, 'www.amazon.com'	Hyperlink
Text	TEXT	Type, 'Amazon'	Hyperlink
Apply	PUSH_BUTTON	Click	Hyperlink
Insert	MENU	Click	LibreOffice Writer
Format	MENU	Click	LibreOffice Writer
Change Case	MENU	Click	LibreOffice Writer
Hyperlink	MENU_ITEM	Click	LibreOffice Writer
Hyperlink	MENU_ITEM	Keyboard Access	LibreOffice Writer
Hyperlink	TOGGLE_BUTTON	Click	LibreOffice Writer
Close	PUSH_BUTTON	Click	Hyperlink
System Interaction	Expand	Unrestricted Focus	Terminal

Table B.6: Rules for Insert Hyperlink task

Rule	Events
Exclusion 1	UPPERCASE, MENU_ITEM, Click, LibreOffice Writer UPPERCASE, MENU_ITEM, Keyboard Access, LibreOffice Writer UPPERCASE, PUSH_BUTTON, Click, LibreOffice Writer
Exclusion 2	Hyperlink, MENU_ITEM, Click, LibreOffice Writer Hyperlink, MENU_ITEM, Keyboard Access, LibreOffice Writer Hyperlink, TOGGLE_BUTTON, Click, LibreOffice Writer
Order 1	Hyperlink, MENU_ITEM, Click, LibreOffice Writer Hyperlink, MENU_ITEM, Keyboard Access, LibreOffice Writer Hyperlink, TOGGLE_BUTTON, Click, LibreOffice Writer
	UPPERCASE, MENU_ITEM, Click, LibreOffice Writer UPPERCASE, MENU_ITEM, Keyboard Access, LibreOffice Writer UPPERCASE, PUSH_BUTTON, Click, LibreOffice Writer
Order 2	Target, TEXT, Type, 'www.amazon.com', Hyperlink Text, TEXT, Type, 'Amazon', Hyperlink
	Apply, PUSH_BUTTON, Click, Hyperlink
	Close, PUSH_BUTTON, Click, Hyperlink
Required	Target, TEXT, Type, 'www.amazon.com', Hyperlink Text, TEXT, Type, 'Amazon', Hyperlink Apply, PUSH_BUTTON, Click, Hyperlink Close, PUSH_BUTTON, Click, Hyperlink

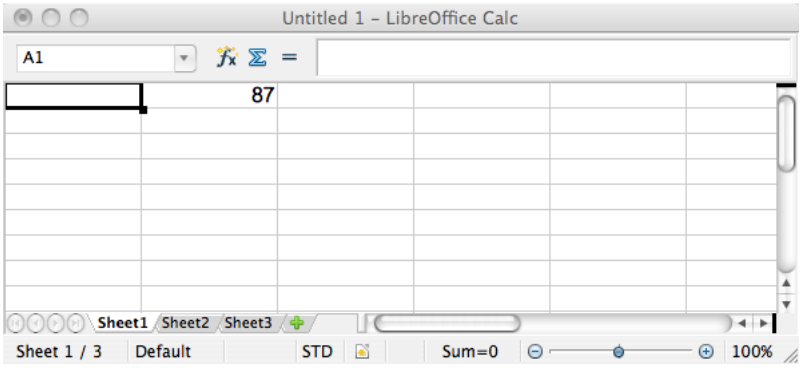


Figure B.3: Result of performing Absolute Value task

Table B.7: Goals, Approaches, and Methods for Absolute Value task

Goal	Approaches	Methods
Open the 'Function Wizard' window.	(a) Menu (b) Keyboard (c) Toolbar	(a) Insert→Function... (b) Type 'Alt-IF' (c) Function Wizard
Click 'Next' button.		
Type '-87' in Number text box.		
Click 'OK' to close the window.		
Shift the cell to the right.	(a) Menu (b) Keyboard (c) Toolbar	(a) Insert→Cells..., Click 'Shift Cells Right', Click 'OK' (b) Type 'Alt-IC', Click 'Shift Cells Right', Click 'OK' (c) Insert Cells Right
Turn off Column & Row Headers	(a) Menu (b) Keyboard	(a) View→Column & Row Headers (b) Type 'Alt-VO'

Table B.8: Event Tuples for Absolute Value task

Title	Class	Action	Window
Cell A1	TABLE_CELL	Click	LibreOffice Calc
Insert Cells Right	PUSH_BUTTON	Click	LibreOffice Calc
Shift Cells Right	RADIO_BUTTON	Click	Insert Cells
Column & Row Headers	MENU_ITEM	Click	LibreOffice Calc
Column & Row Headers	MENU_ITEM	Keyboard Access	LibreOffice Calc
Next	PUSH_BUTTON	Click	Function Wizard
Number	TEXT	Type, '-87'	Function Wizard
Insert	MENU	Click	LibreOffice Calc
View	MENU	Click	LibreOffice Calc
Function...	MENU_ITEM	Click	LibreOffice Calc
Function...	MENU_ITEM	Keyboard Access	LibreOffice Calc
Function Wizard	PUSH_BUTTON	Click	LibreOffice Calc
Cells...	MENU_ITEM	Click	LibreOffice Calc
Cells...	MENU_ITEM	Keyboard Access	LibreOffice Calc
OK	PUSH_BUTTON	Click	Insert Cells
OK	PUSH_BUTTON	Click	Function Wizard
System Interaction	Expand	Restricted Focus	Terminal

Table B.9: Rules for Absolute Value task

Rule	Events
Exclusion 1	Insert Cells Right, PUSH_BUTTON, Click, LibreOffice Calc Shift Cells Right, RADIO_BUTTON, Click, Insert Cells
Exclusion 2	Column & Row Headers, MENU_ITEM, Click, LibreOffice Calc Column & Row Headers, MENU_ITEM, Keyboard Access, LibreOffice Calc
Exclusion 3	Function..., MENU_ITEM, Click, LibreOffice Calc Function..., MENU_ITEM, Keyboard Access, LibreOffice Calc Function Wizard, PUSH_BUTTON, Click, LibreOffice Calc
Order 1	Cell A1, TABLE_CELL, Click, LibreOffice Calc
	Function..., MENU_ITEM, Click, LibreOffice Calc Function..., MENU_ITEM, Keyboard Access, LibreOffice Calc Function Wizard, PUSH_BUTTON, Click, LibreOffice Calc
	Insert Cells Right, PUSH_BUTTON, Click, LibreOffice Calc Shift Cells Right, RADIO_BUTTON, Click, Insert Cells
Order 2	Next, PUSH_BUTTON, Click, Function Wizard
	Number, TEXT, Type,'-87', Function Wizard
Required	Next, PUSH_BUTTON, Click, Function Wizard Number, TEXT, Type,'-87', Function Wizard Cell A1, TABLE_CELL, Click, LibreOffice Calc
Repeat	Insert, MENU, Click, LibreOffice Calc

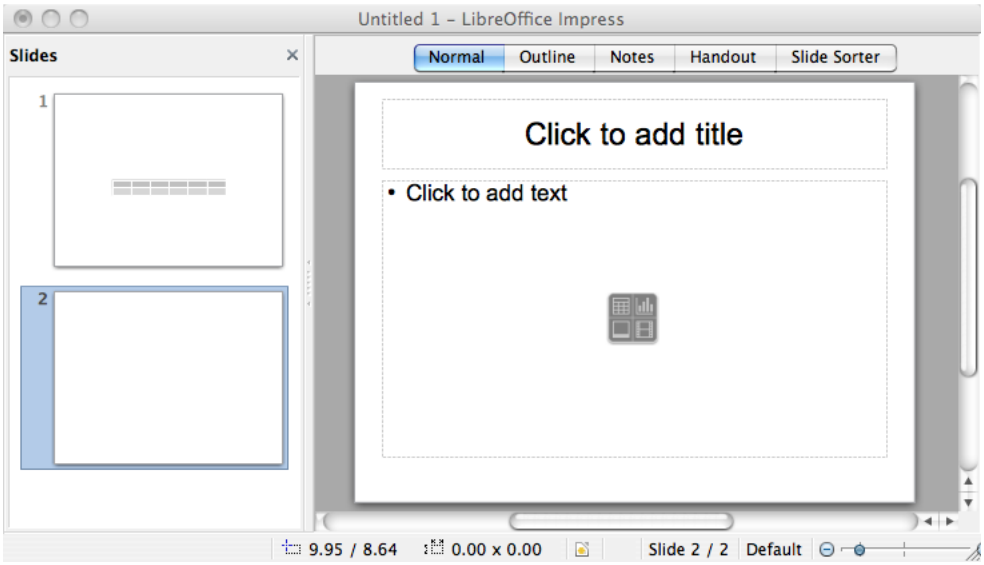


Figure B.4: Result of performing Insert Table task

Table B.10: Goals, Approaches, and Methods for Insert Table Task

Goals	Approaches	Methods
Open the 'Table' window.	(a) Menu (b) Toolbar	(a) Insert→Table (b) Table
Type '6' in Columns text box.		
Click 'OK' to close the Table window.		
Add a new slide.	(a) Menu (b) Keyboard (c) Toolbar	(a) Insert → Slide (b) Type 'Alt-IE' (c) New Slide
Hide the Task Pane	(a) Menu (b) Keyboard	(a) View→Task Pane (b) Type 'Alt-VK'

Table B.11: Event tuples for Insert Table task

Title	Class	Action	Window
Task Pane	MENU_ITEM	Click	LibreOffice Impress
Task Pane	MENU_ITEM	Keyboard Access	LibreOffice Impress
Number of Columns	TEXT	Type,'6'	Table
Slide	MENU_ITEM	Click	LibreOffice Impress
Slide	PUSH_BUTTON	Click	LibreOffice Impress
Slide	MENU_ITEM	Keyboard Access	LibreOffice Impress
Insert	MENU	Click	LibreOffice Impress
View	MENU	Click	LibreOffice Impress
Table	MENU_ITEM	Click	LibreOffice Impress
Table	PUSH_BUTTON	Click	LibreOffice Impress
OK	PUSH_BUTTON	Click	Table

System Interaction	Expand	Restricted Focus	Terminal

Table B.12: Rules for Insert Table task

Rule	Events
Exclusion 1	Task Pane, MENU_ITEM, Click, LibreOffice Impress Task Pane, MENU_ITEM, Keyboard Access, LibreOffice Impress
Exclusion 2	Slide, MENU_ITEM, Click, LibreOffice Impress Slide, PUSH_BUTTON, Click, LibreOffice Impress Slide, MENU_ITEM, Keyboard Access, LibreOffice Impress
Exclusion 3	Table, MENU_ITEM, Click, LibreOffice Impress Table, PUSH_BUTTON, Click, LibreOffice Impress
Order 1	Table, MENU_ITEM, Click, LibreOffice Impress Table, PUSH_BUTTON, Click, LibreOffice Impress
	Slide, MENU_ITEM, Click, LibreOffice Impress Slide, PUSH_BUTTON, Click, LibreOffice Impress Slide, MENU_ITEM, Keyboard Access, LibreOffice Impress
Required	Number of Columns,TEXT, Type,‘6’, Table OK, PUSH_BUTTON, Click, Table
Repeat	Insert, MENU, Click, LibreOffice Impress

Bibliography

[1] E. Abdulin. Using the Keystroke-Level Model for designing user-interfaces on middle-sized touchscreens. In *Proceedings of the SIGCHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI 11, pages 673–686, New York, NY, USA, 2011. ACM.

[2] A. Agarwal and M. Prabaker. Building on the usability study: Two explorations on how to better understand an interface. In *New Trends in Human-Computer Interaction*, volume 5610 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin / Heidelberg, 2009.

[3] R. St. Amant and M. O. Riedl. A perception/action substrate for cognitive modeling in HCI. *International Journal of Human-Computer Studies*, 55(1):15 – 39, 2001.

[4] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036, 2004.

[5] J. R. Anderson and C. Lebiere. *The atomic components of thought*. Lawrence Erlbaum Associates Inc., 1998.

[6] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[7] R. K. Bellamy, B. E. John, and S. Kogan. Deploying CogTool: Integrating quantitative usability assessment into real-world software development. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE 11, pages 691–700, New York, NY, USA, 2011. ACM.

[8] F. Belli. Finite state testing and analysis of graphical user interfaces. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ISSRE 01, pages 34–43. IEEE, 2001.

[9] S. K. Card, T. P. Moran, and A. Newell. The Keystroke-Level Model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, 1980.

[10] S. K. Card, A. Newell, and T. P. Moran. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.

[11] T. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI 10, pages 1535–1544, New York, NY, USA, 2010. ACM.

[12] E. H. Chi, A. Rosien, G. Supattanasiri, A. Williams, C. Royer, C. Chow, E. Robles, B. Dalal, J. Chen, and S. Cousins. The Bloodhound Project: Automating discovery of web usability issues using the InfoScent simulator. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI 03, pages 505–512, 2003.

[13] W. T. Fu and P. Pirolli. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. *Human–Computer Interaction*, 22(4):355–412, 2007.

[14] W. D. Gray, B. E. John, and M. E. Atwood. Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human Computer Interaction*, 8(3):237–309, 1993.

[15] M. Grechanik, Q. Xie, and C. Fu. Creating GUI testing tools using accessibility technologies. In *International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 09, pages 243–250. IEEE, 2009.

[16] The Java Accessibility API. http://java.sun.com/javase/technologies/accessibility/index.jsp, 2012.

[17] B. E. John and D. E. Kieras. Using GOMS for user-interface design and evaluation: Which technique? *ACM Transactions on Computer-Human Interaction*, 3(4):287–319, 1996.

[18] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger. Predictive human performance modeling made easy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI 04, pages 455–462. ACM, 2004.

[19] B. E. John and S. Suzuki. Toward cognitive modeling for predicting usability. In *New Trends in Human-Computer Interaction*, volume 5610 of *Lecture Notes in Computer Science*, pages 267–276. Springer Berlin / Heidelberg, 2009.

[20] D. E. Kieras. A guide to GOMS model usability evaluation using GOMSL and GLEAN3, 1999.

[21] A. Knight, G. Pyrzak, and C. Green. When two methods are better than one: Combining user study with cognitive modeling. In *Proceedings of the SIGCHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI 07, pages 1783–1788, New York, NY, USA, 2007. ACM.

[22] P. Li, T. Huynh, M. Reformat, and J. Miller. A practical approach to testing GUI systems. *Empirical Software Engineering*, 12:331–357, 2007.

[23] LibreOffice: The Document Foundation. http://www.libreoffice.org/, 2012.

[24] L. Lu and D. P. Siewiorek. KLEM: A method for predicting user interaction time and system energy consumption during application design. In *The 11th IEEE International Symposium on Wearable Computers*, pages 69 –76, 2007.

[25] L. Luo and B. E. John. Predicting task execution time on handheld devices using the Keystroke-Level Model. In *Proceedings of the SIGCHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI 05, pages 1605–1608, New York, NY, USA, 2005. ACM.

[26] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, University of Pittsburgh, 2001.

[27] A. M. Memon. GUI testing: Pitfalls and processes. *Computer*, 35(8):87–88, August 2002.

[28] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[29] A. M. Memon. TerpOffice. http://www.cs.umd.edu/ atif/TerpOffice/, 2011.

[30] A. M. Memon. GUITAR - A GUI Testing frAmewoRk . http://guitar.sourceforge.net, 2012.

[31] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE 03. IEEE, 2003.

[32] A. M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, 2001.

[33] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 118–127, New York, NY, USA, 2003. ACM.

[34] A.M. Memon, M.L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 256–267. ACM, 2001.

[35] Microsoft UI Automation. http://msdn.microsoft.com/en-us/library/ms747327.aspx, 2012.

[36] J. Monkiewicz. CAD's next-generation user-interface. *Computer-Aided Engineering*, pages 55–56, November 1992.

[37] Microsoft Office. http://office.microsoft.com/, 2012.

[38] NeoOffice. http://www.neooffice.org/neojava/en/index.php, 2012.

[39] The OpenOffice Productivity Suite. http://www.openoffice.org/, 2012.

[40] The UNO Accessibility API. http://www.openoffice.org/ui/accessibility/ unoapi.html, 2012.

[41] E. Patton and W. Gray. SANLab-CM: A tool for incorporating stochastic operations into activity network modeling. *Behavior Research Methods*, 42:877–883, 2010.

[42] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[43] D. D. Salvucci. Predicting the effects of in-car interfaces on driver behavior using a cognitive architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI 01, pages 120–127, New York, NY, USA, 2001. ACM.

[44] D. D. Salvucci and F. J. Lee. Simple cognitive modeling in a complex cognitive architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI 03, pages 265–272. ACM, 2003.

[45] F. Shull, J. Singer, and D. I. K. Sjoberg. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag, London, United Kingdom, 2008.

[46] K. Stuart, K. William, and J. Betty. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics*, 21(8):601–613, 1978.

[47] A. Swearngin, M. B. Cohen, B. E. John, and R. K. Bellamy. Easing the generation of predictive human performance models from legacy systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, To Appear*, CHI 12. ACM, 2012.

[48] L. Teo, B. E. John, and M. H. Blackmon. CogTool-Explorer: A model of goal-directed user exploration that considers information layout. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, To Appear*, CHI 12. ACM, 2012.

[49] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora. A framework for measurement based performance modeling. In *Proceedings of the 7th International Workshop on Software and Performance*, WOSP 08, pages 55–66, 2008.

[50] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE 00, pages 110–121. IEEE, 2000.

[51] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE 05, pages 293–302, 2005.

[52] X. Yuan and A.M. Memon. Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2010.